Current status of the development of the Unified Solids library

Marek Gayer

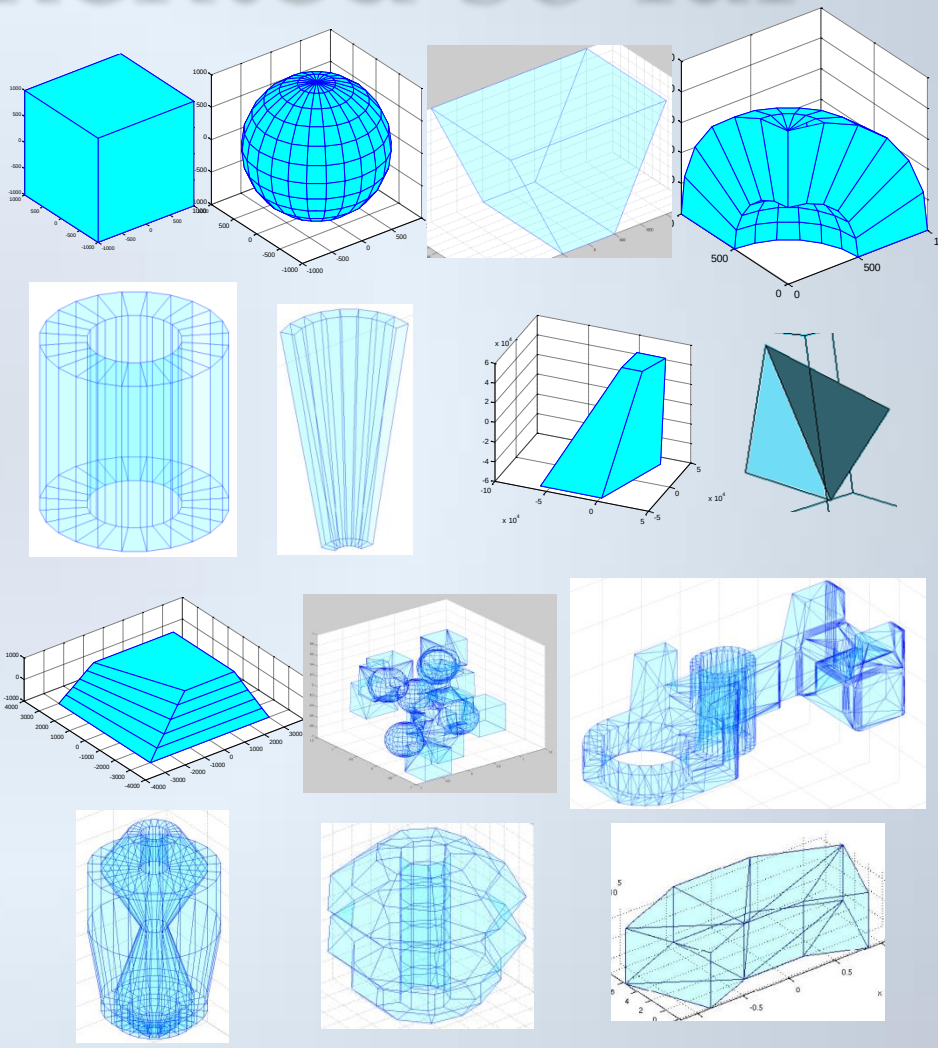
CERN PH/SFT

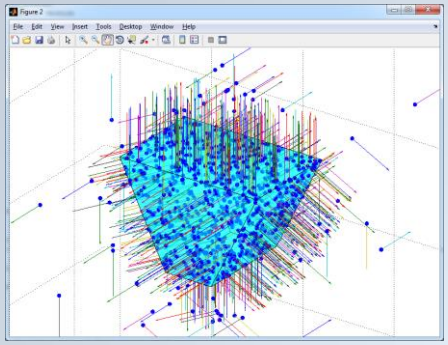
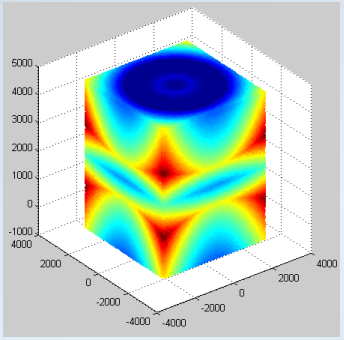
Motivations for a common solids library

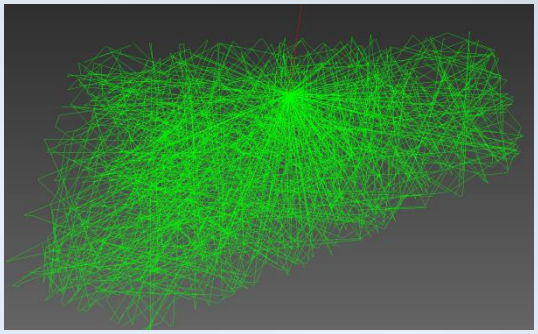
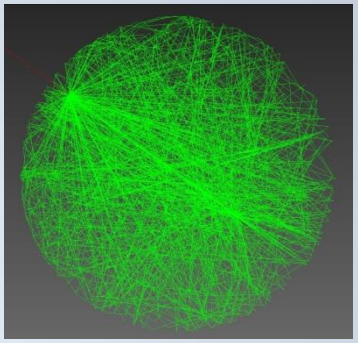
- Optimize and guarantee better long-term maintenance of ROOT and Gean4 solids libraries
  - A rough estimation indicates that about 70-80% of code investment for the geometry modeler concerns solids, to guarantee the required precision and efficiency in a huge variety of combinations
- Create a single high quality library to replace solid libraries in Geant4 and ROOT
  - Starting from what exists today in Geant4 and ROOT
  - Adopt a single type for each shape
  - Significantly optimize (Multi-Union, Tessellated Solid, Polyhedra, Polycone)
  - Reach complete conformance to GDML solids schema
- Create extensive testing suite

Solids implemented so far

- Box
- Orb
- Trapezoid
- Sphere (+ sphere section)
- Tube (+ cylindrical section)
- Cone (+ conical section)
- Generic trapezoid
- Tetrahedron
- Arbitrary Trapezoid (ongoing)
- **Multi-Union**
- **Tessellated Solid**
- **Polycone**
- **Polyhedra**
- Extruded solid (ongoing)



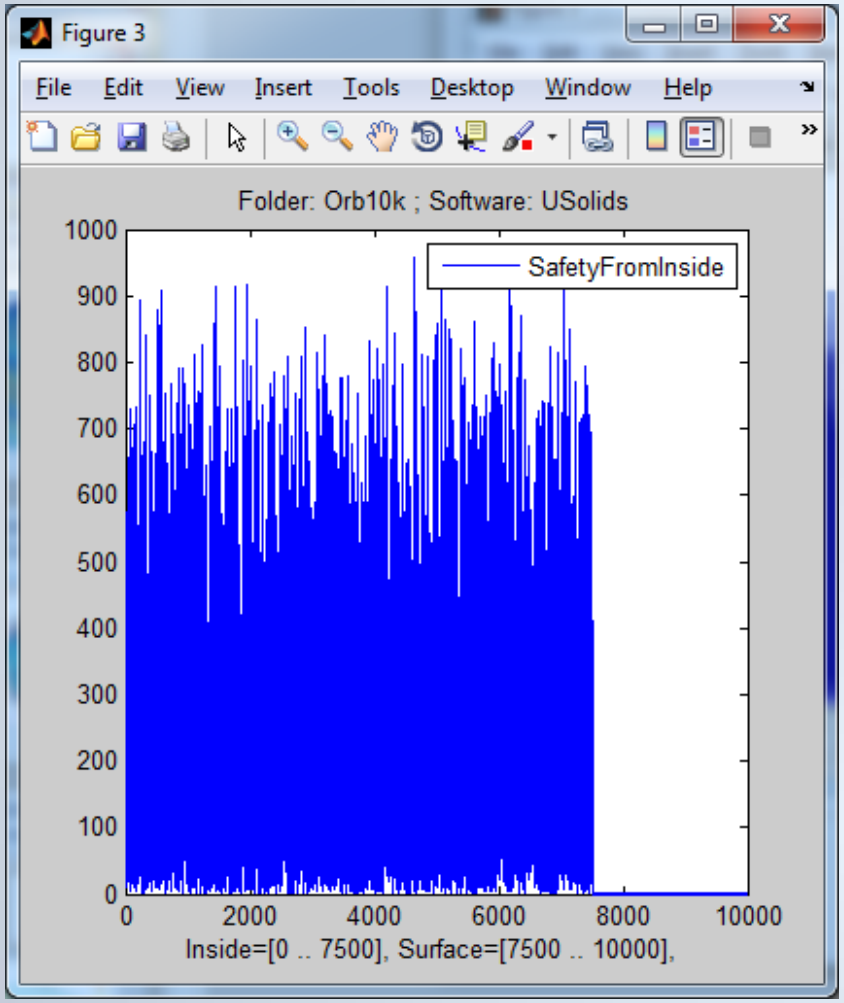
Testing Suite

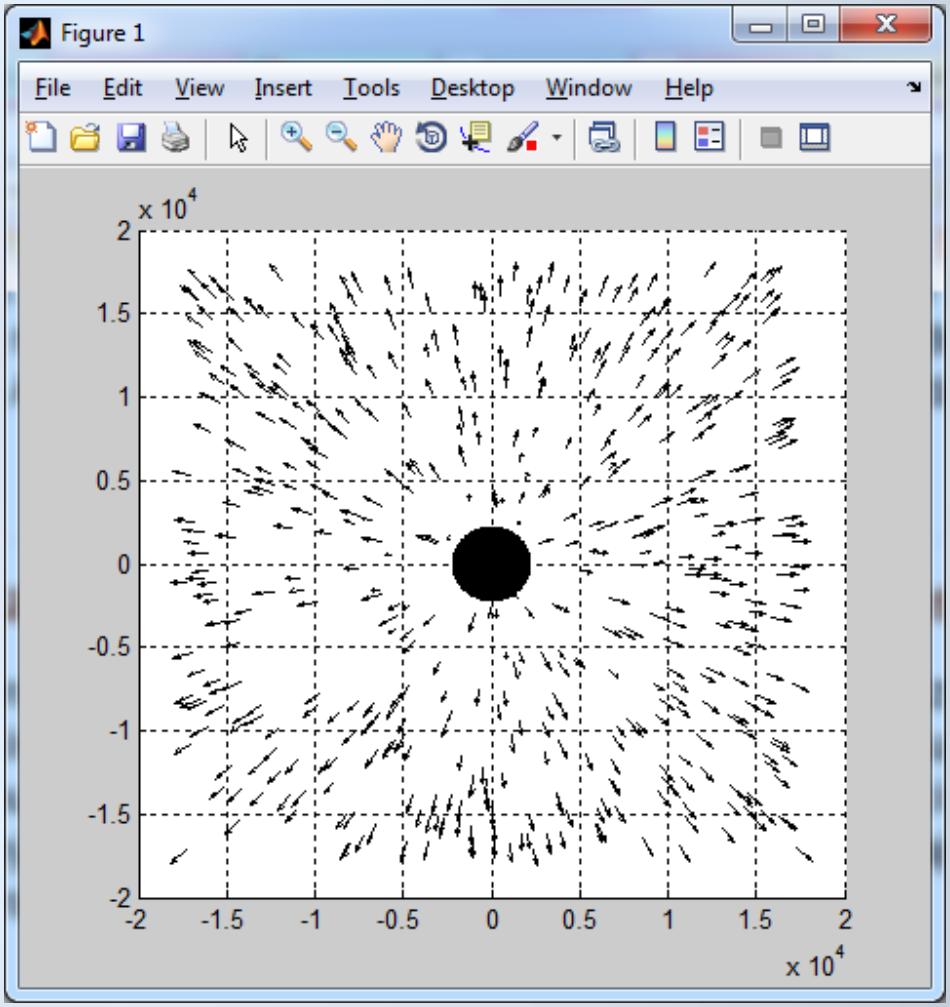
- Unified Solid Batch Test
- Optical Escape
- Specialized tests (e.g. quick performance scalability test for multi-union)

Unified Solid Batch Test

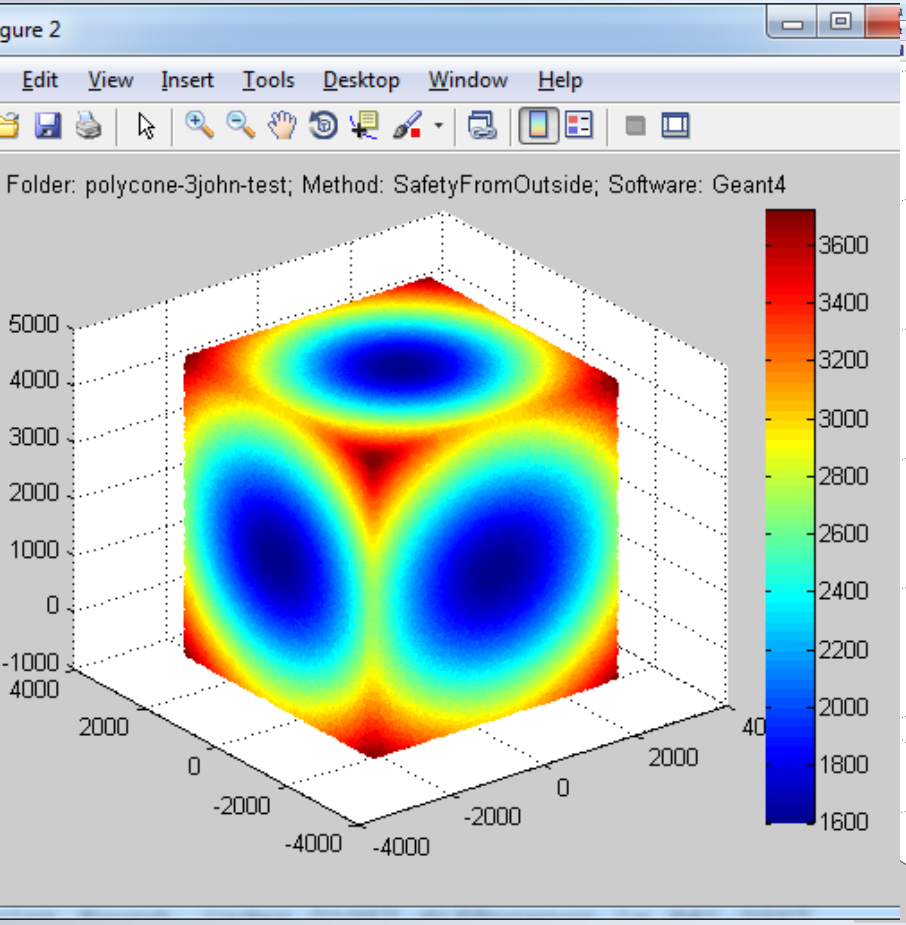
- New powerful diagnosis tool for testing solids of ROOT, Geant4 and Unified Solids library designed for transition to Unified Solids
- Compares performance and output values from different codes
- Helps us to make sure we have similar or better performance in each method and different test cases
- Support for batch configuration and execution of tests
- Useful to detect, report and fix numerous bugs or suspect return values in Geant4, ROOT and Unified Solids
- Two phases
  - Sampling phase (generation of data sets, implemented in C++
  - Analysis phase (data post-processing, implemented as MATLAB scripts)

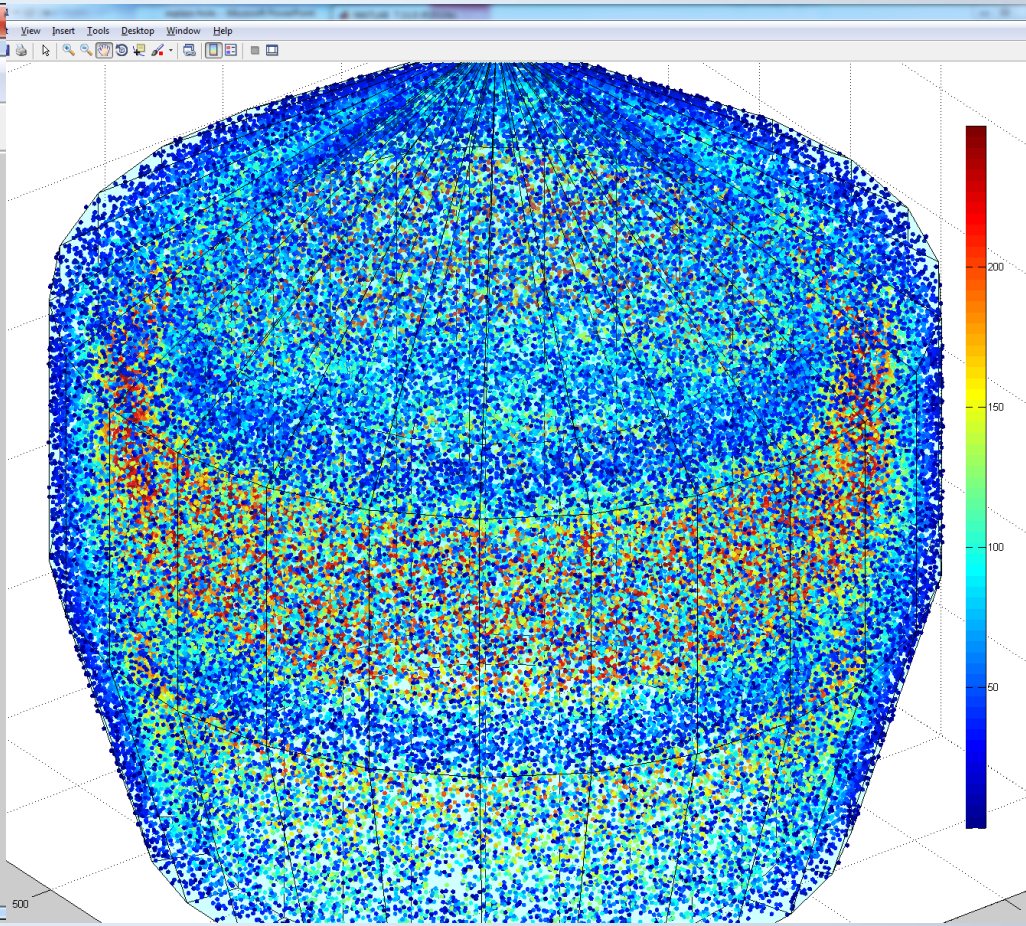
Visualization of scalar and vector data sets



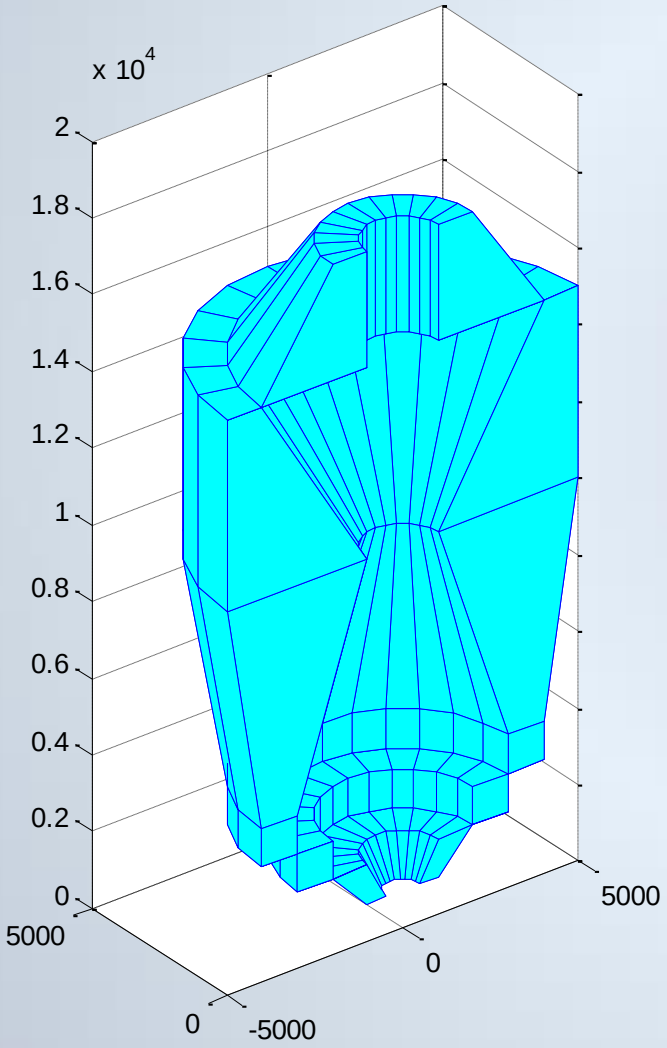


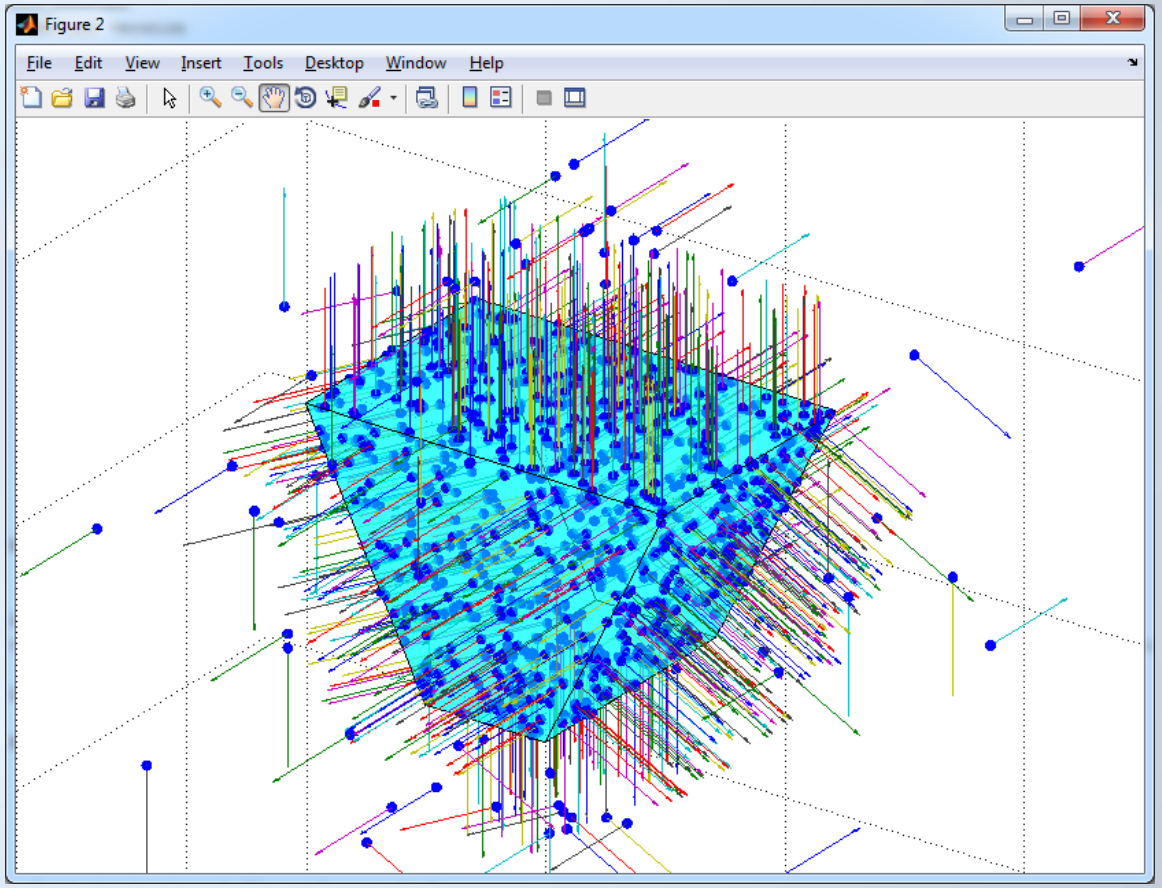
3D plots allowing to overview data sets



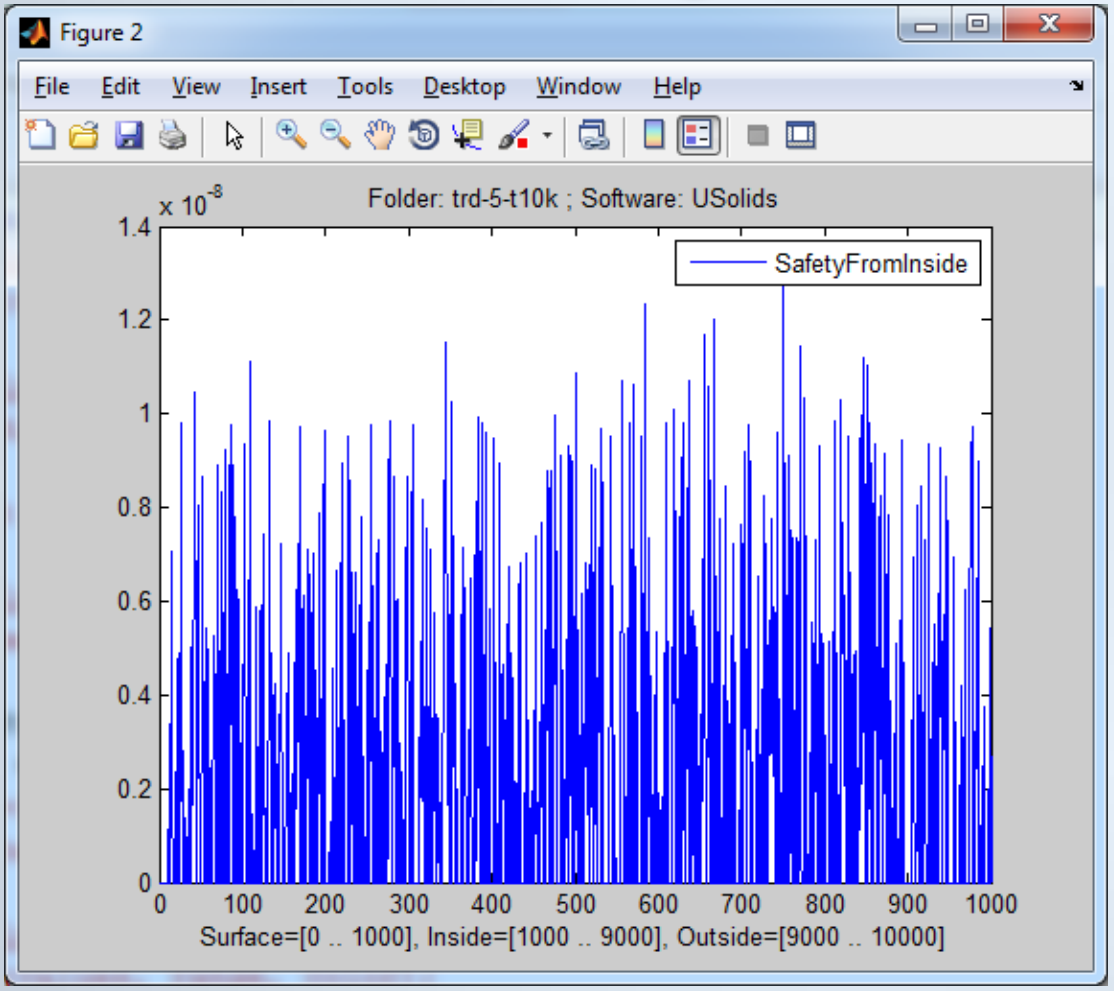


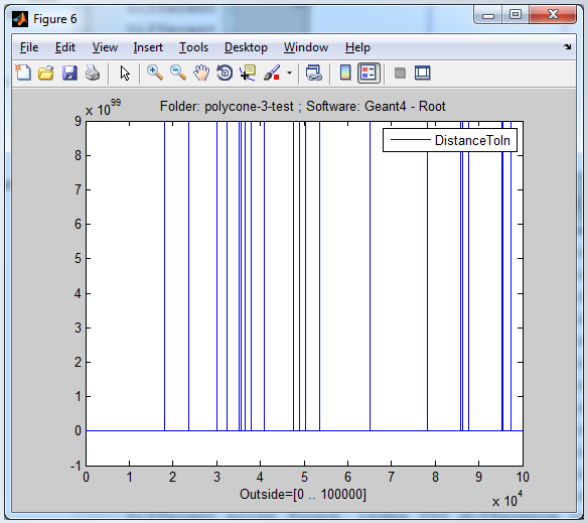
3D visualization of investigated shapes

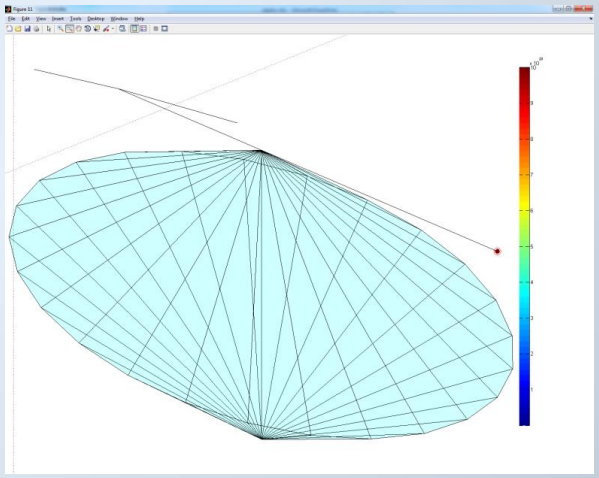




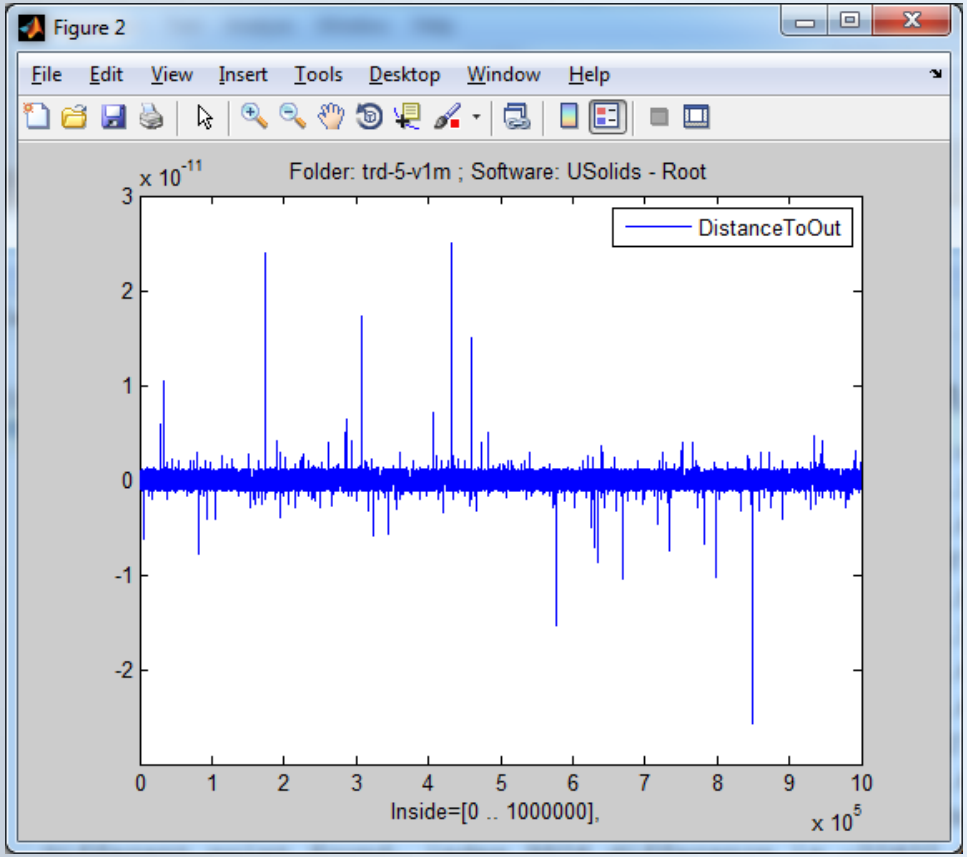
Support for regions of data, focusing on sub-parts

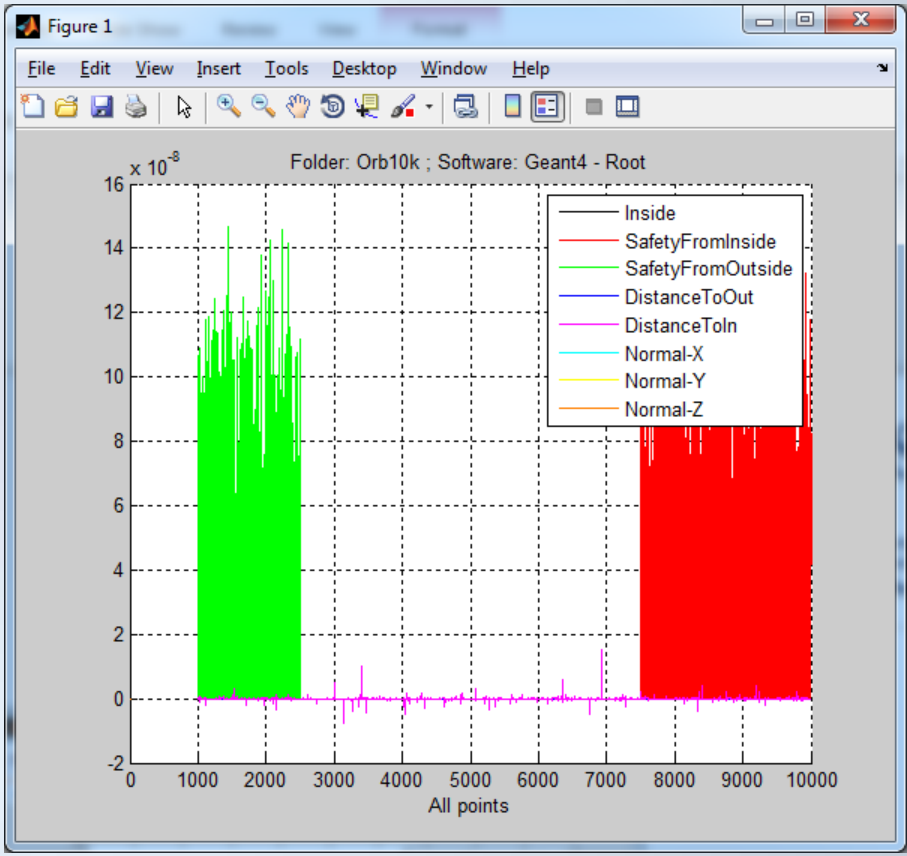




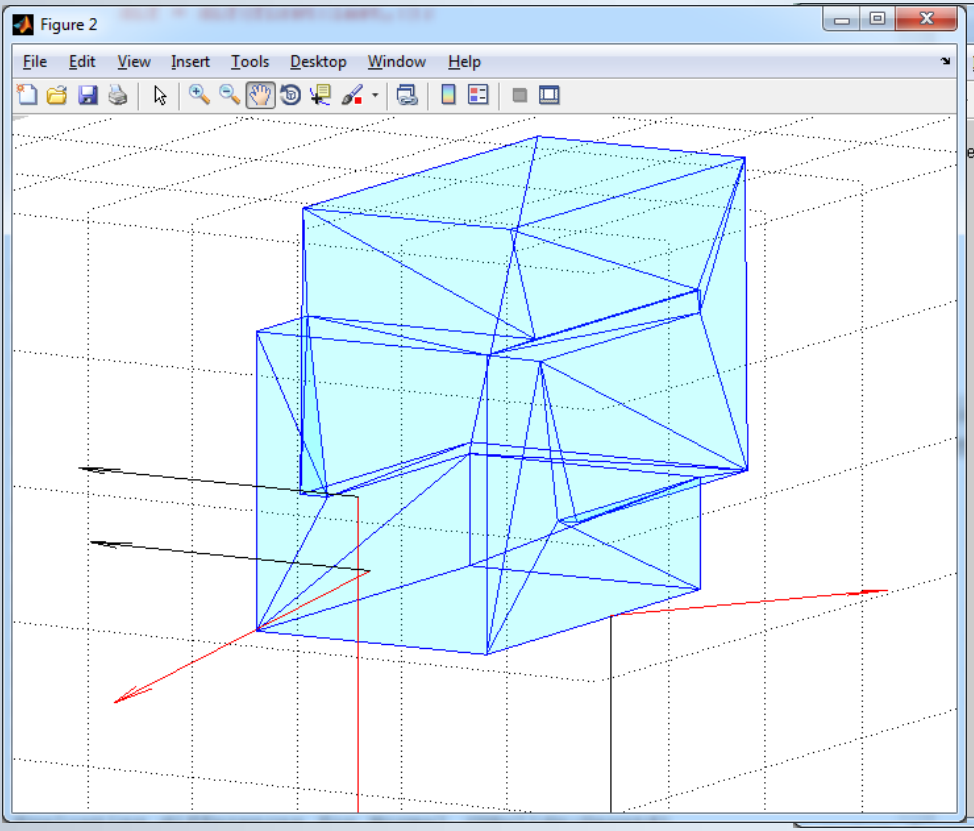


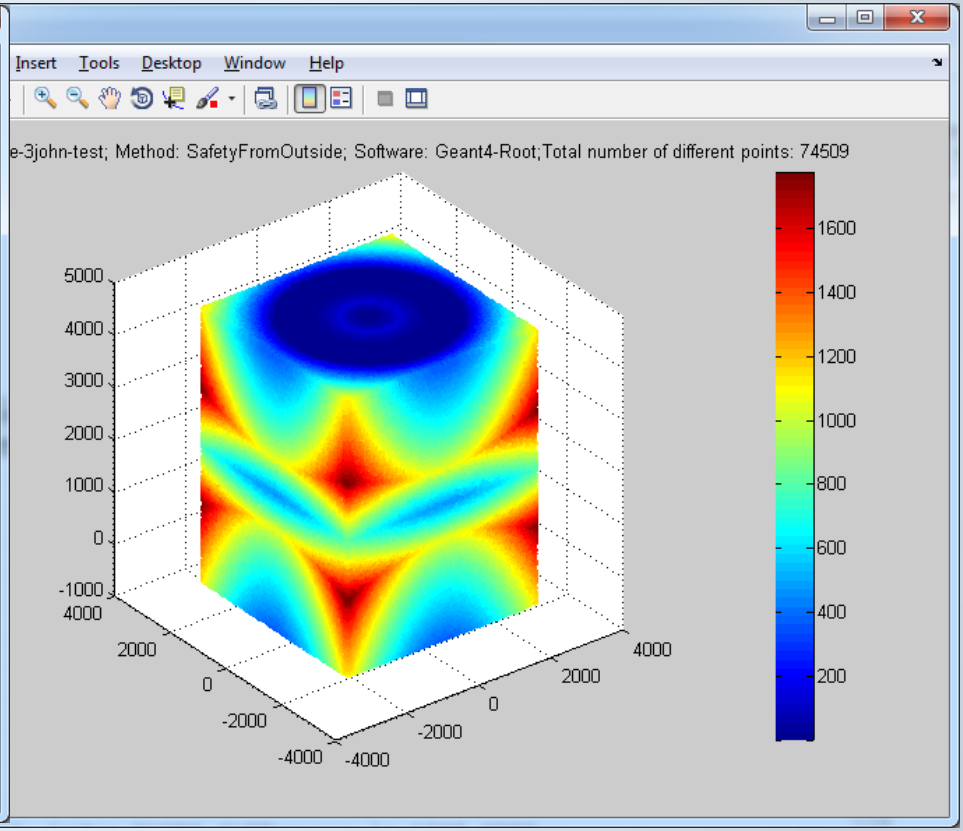
Visual analysis of differences



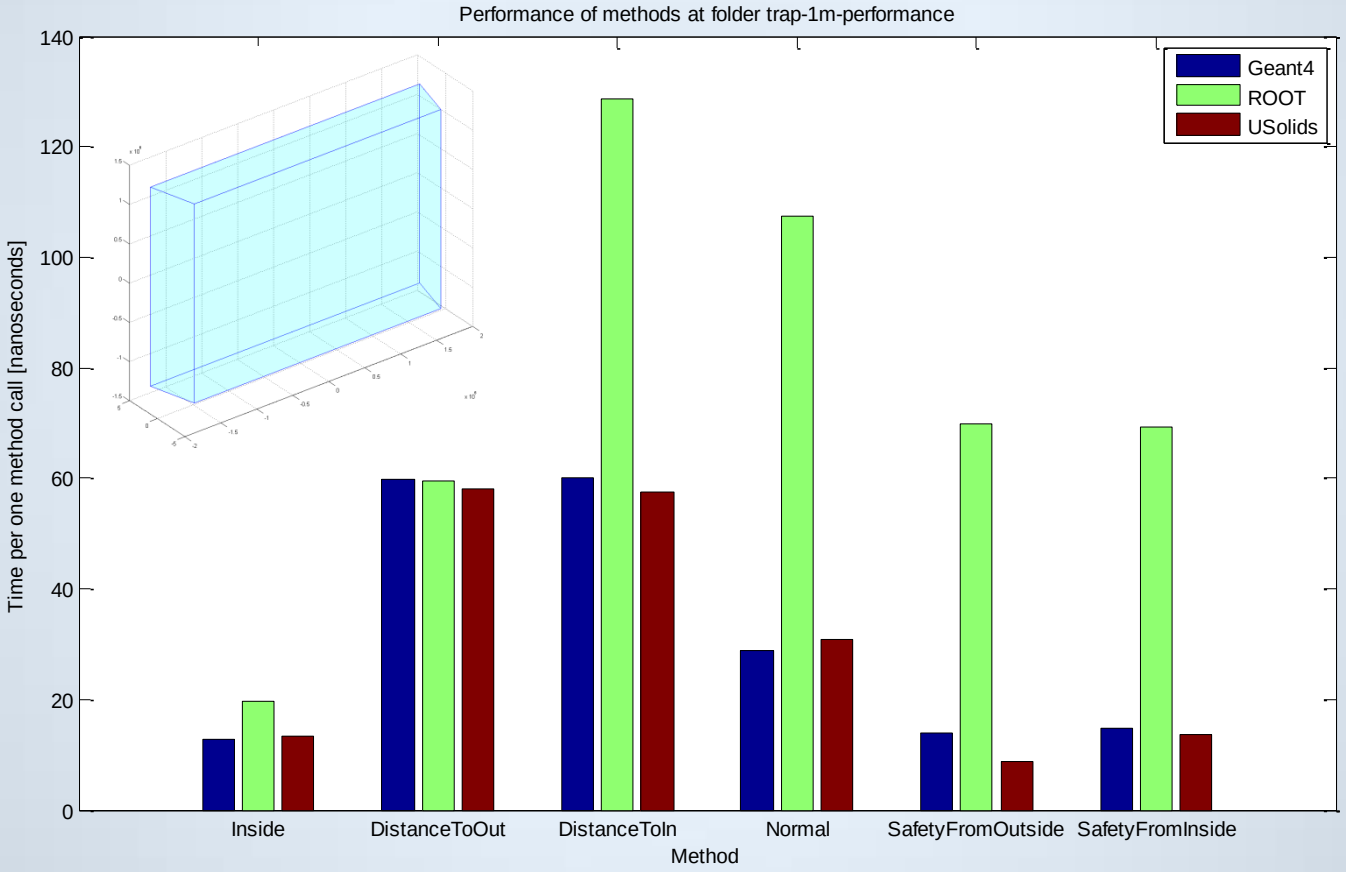


Visual analysis of differences in 3D

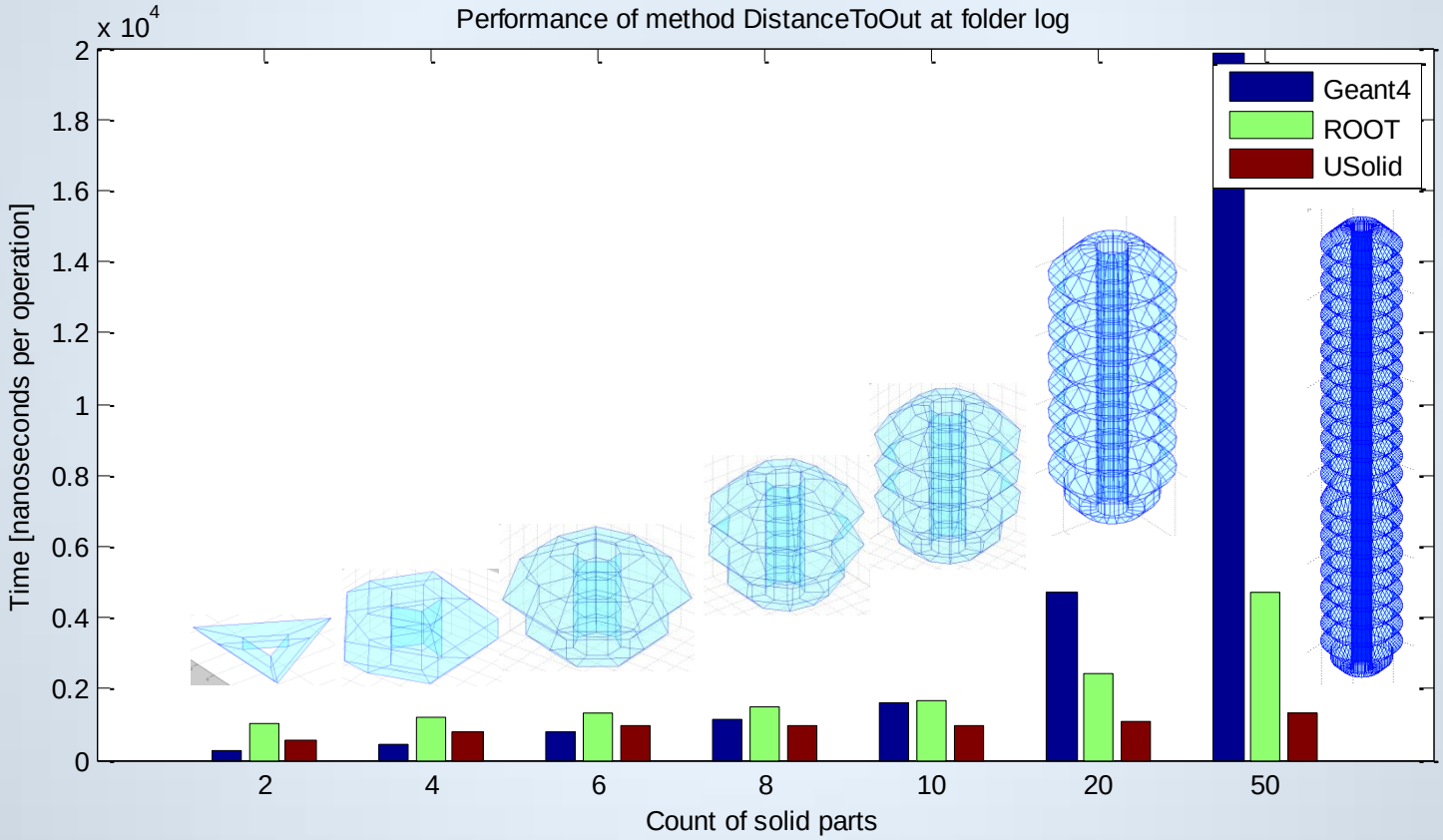




Graphs with comparison of performance



Visualization of scalability performance for specific solids

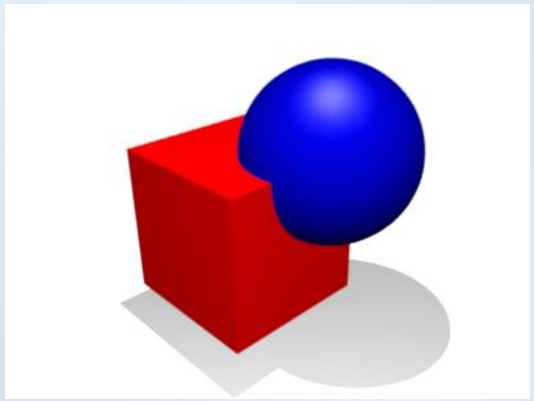


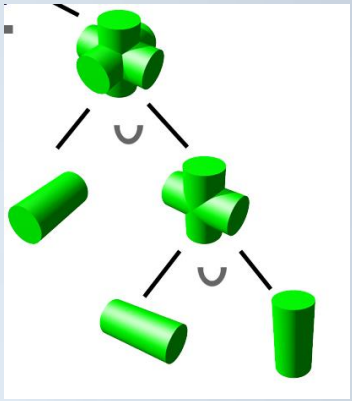
New Multi-Union solid

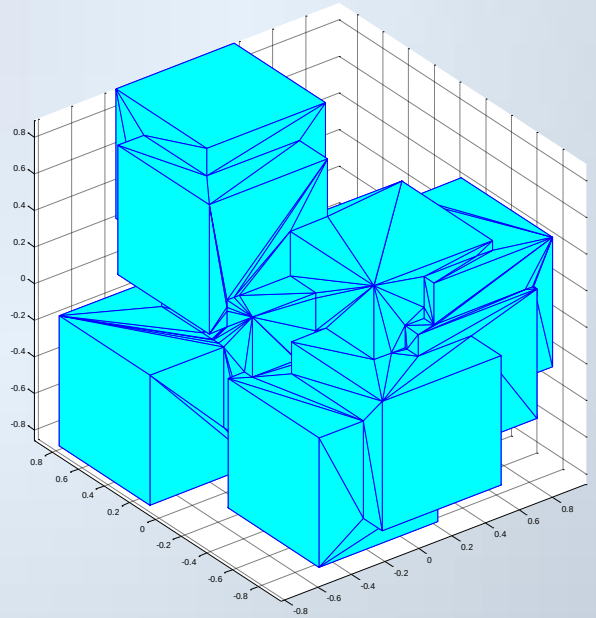
...

Multi-Union solid

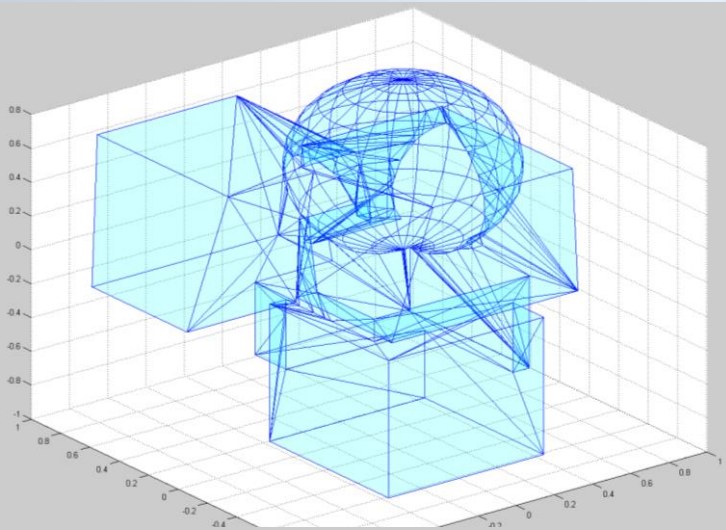
- We implemented a new solid as a union of many solids using voxelization technique to optimize speed and scalability
  - 3D space partition for fast localization of components
  - Aiming for a log(n) scalability, unlike the Geant4 Boolean solid
- Useful for complex composites made of many solids

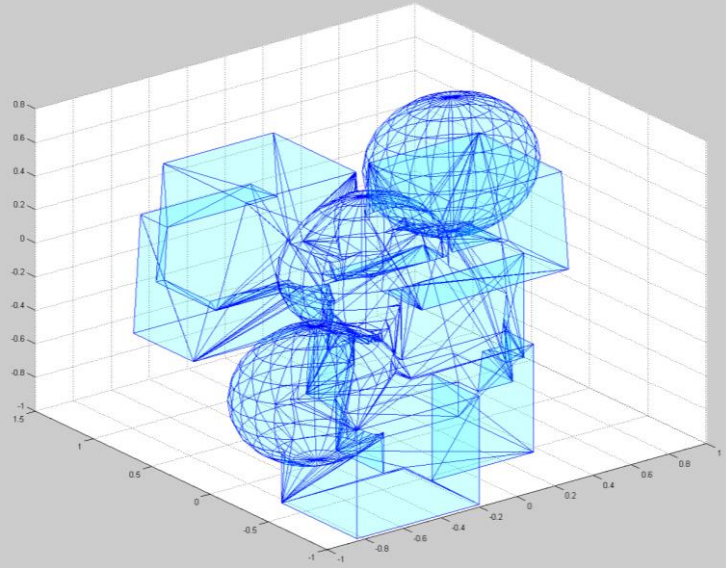




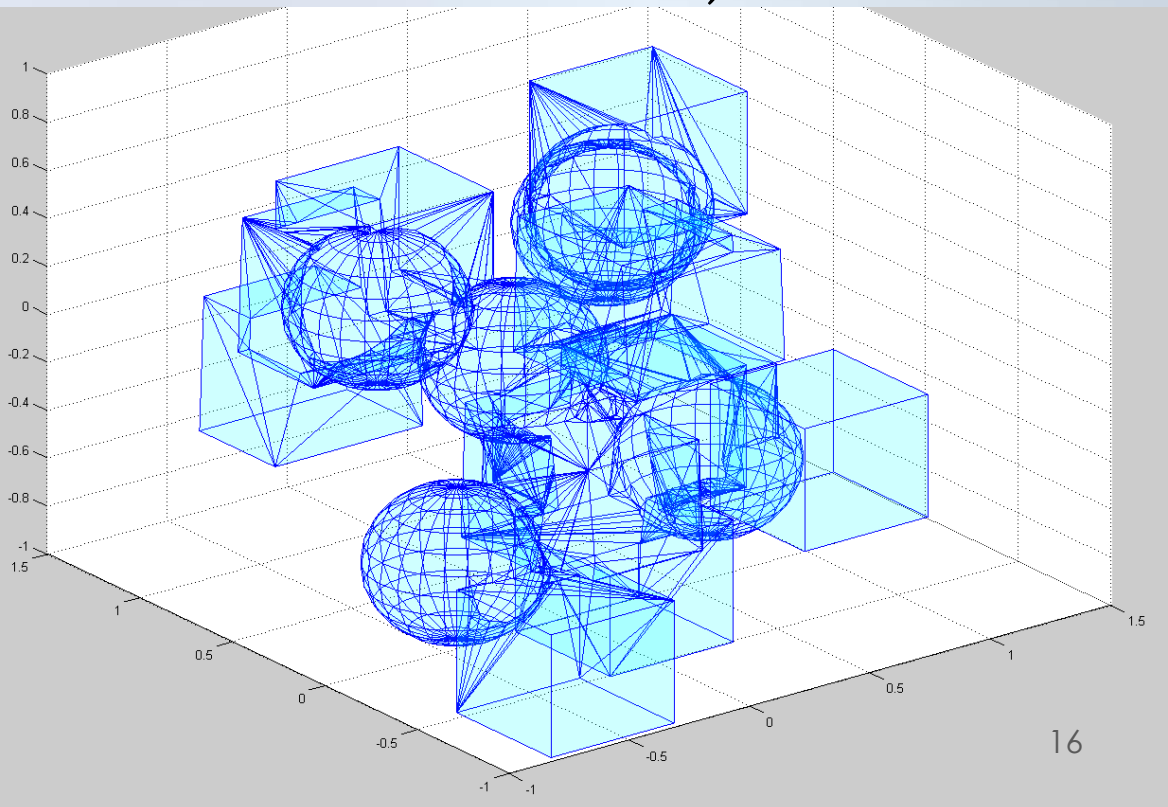


Test multi-union solid for scalability measurements

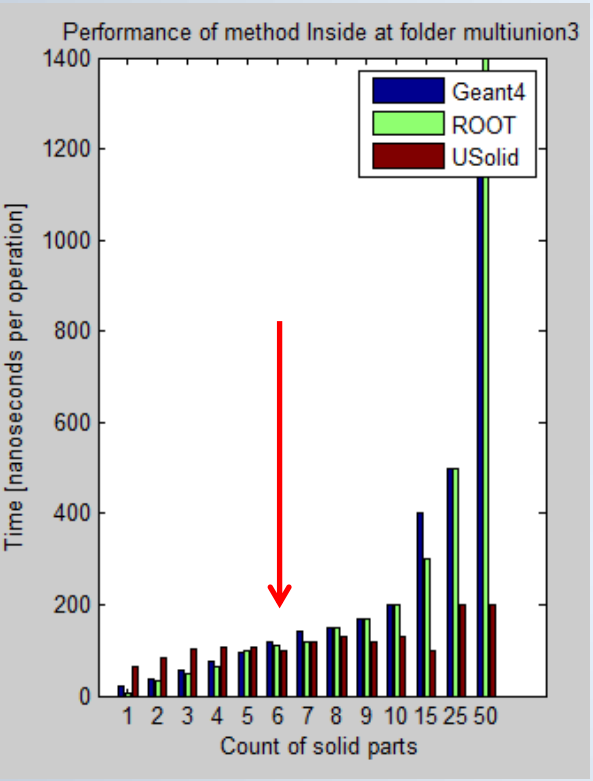


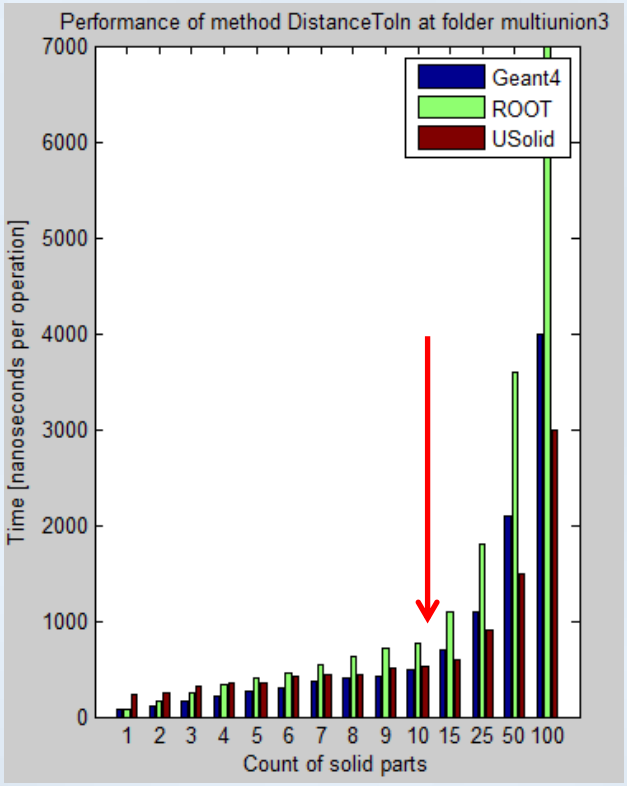


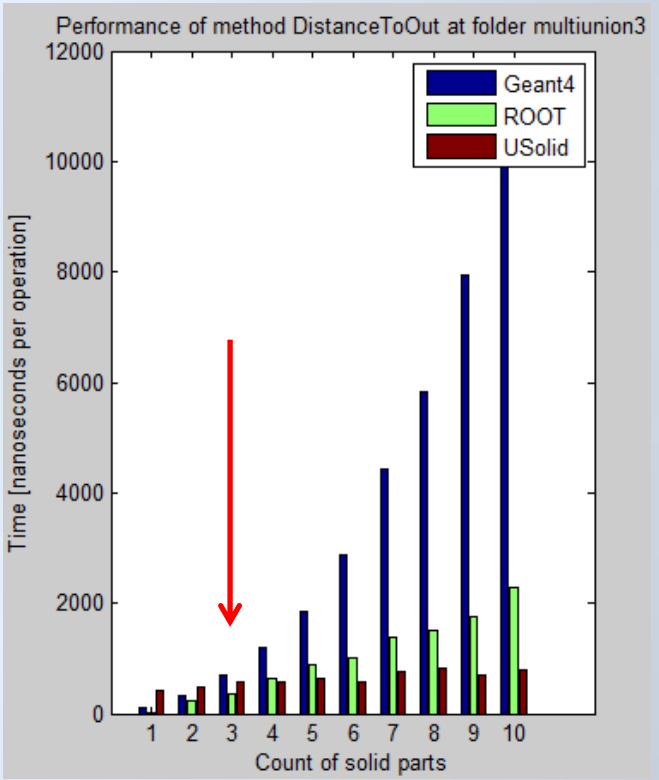
Union of random boxes, orbs and trapezoids (on pictures with 5, 10, 20 solids)



The most performance critical methods



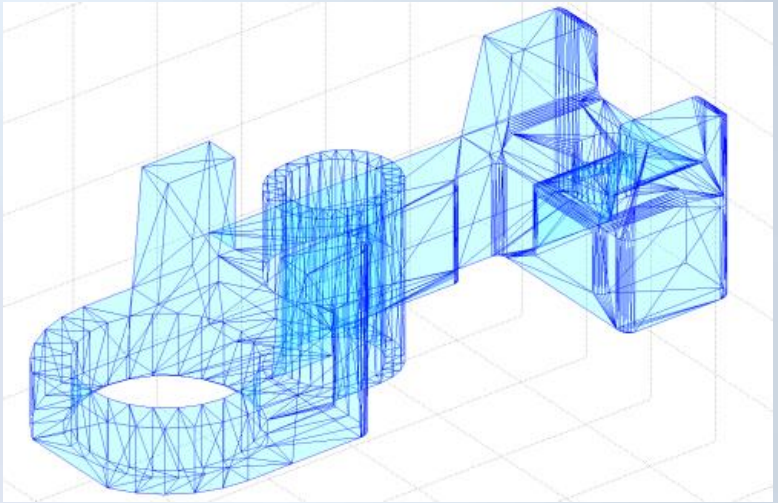


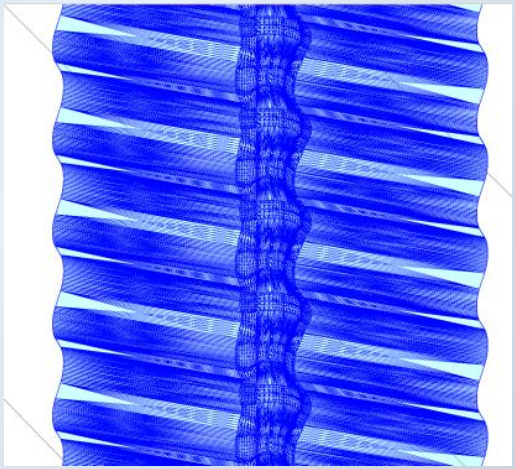


Fast Tessellated Solid

Fast Tessellated Solid

- Made from connected triangular and quadrangular facets forming solid
- Old implementation was slow, no spatial optimization
- We use spatial division of facets into 3D grid forming voxels
- Voxelization is based on:
  - Bitmasks and logical *and* operations during initialization
  - **NEW: Bitmasks are not used at runtime, only pre-calculated lists of facets candidates are used during runtime**
- **Speedup factor ~10x** additional improvement since the previous Geant4 collaboration meeting

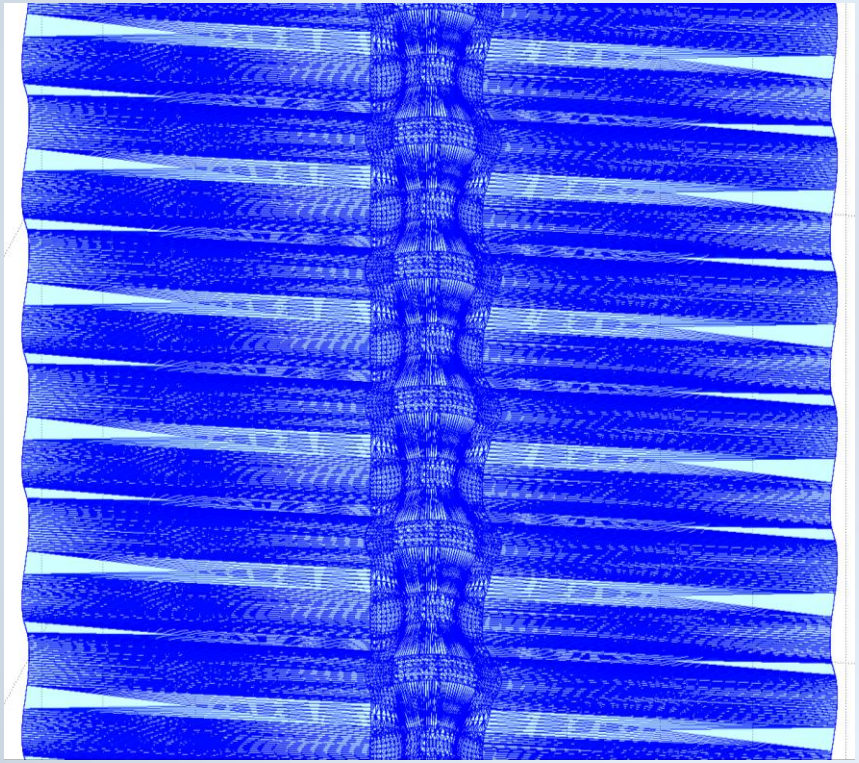




Fast Tessellated Solid

- Old Tessellated Solid had also several weak parts of algorithm, used at initialization with n^2 complexity
  - This sometimes caused very huge delays when loading (e.g. in case of foil with 164k faces)
  - Rewrote to have $n \cdot \log n$ complexity
  - Loading the solid is much faster. What before took minutes, now takes seconds
- Backported in autumn 2012 also to Geant4 **9.6**
- By improving design of classes and removing inefficiencies, total memory save is about ~50%,
  - Voxelization has overhead vs. original G4TS, which reduces to a total memory save of ~25%

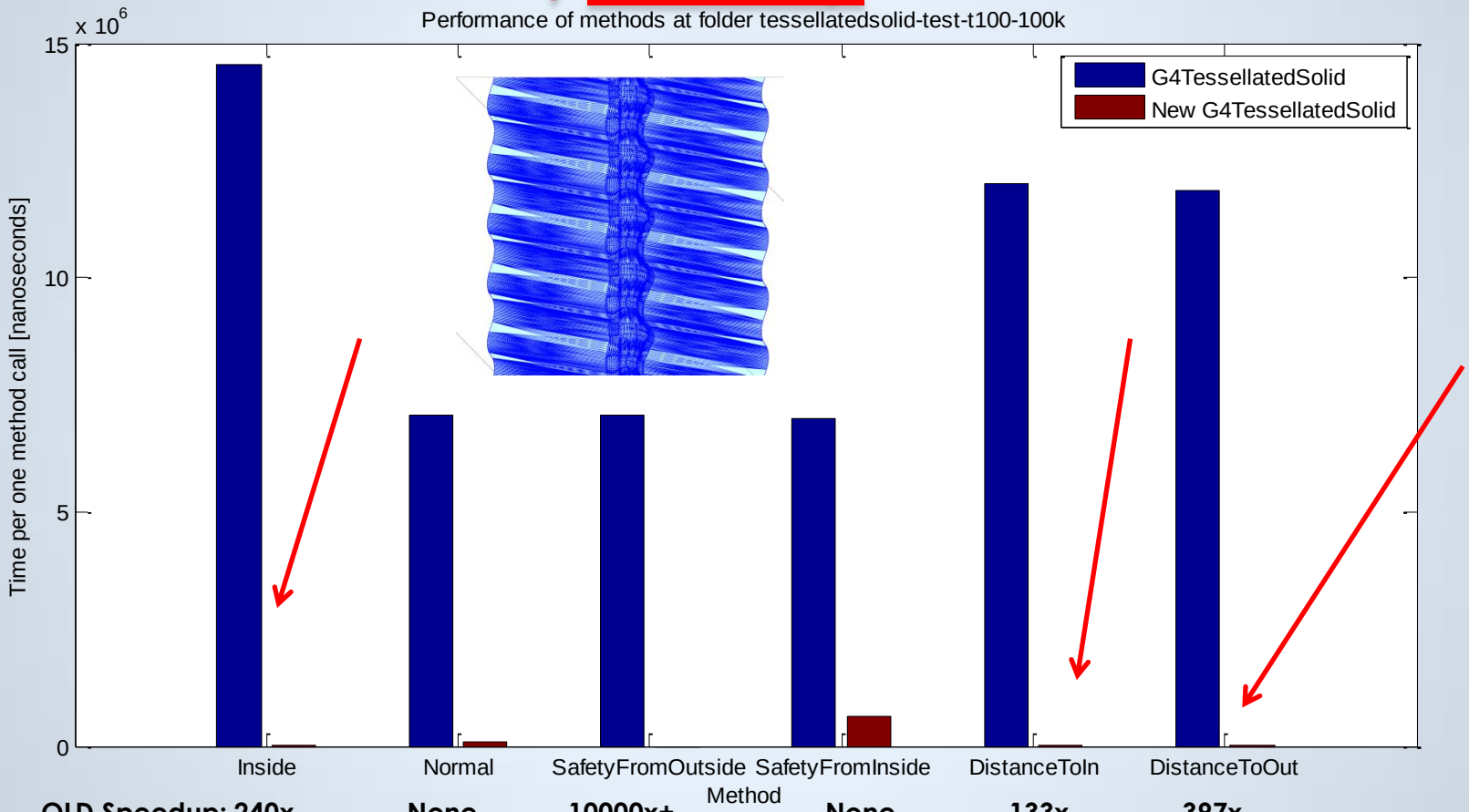
LHCb VELO foil benchmark



Method	Speedup
Inside	**2423x**
DistanceToIn	**1334x**
DistanceToOut	**1976x**

Information	Value
Number of facets	**164.149**
Number of voxels	**158.928**
Memory saved compared with original Geant4	**22% (51MB)**

Performance – 164k/SCL5 with 10k/100k/1M voxels



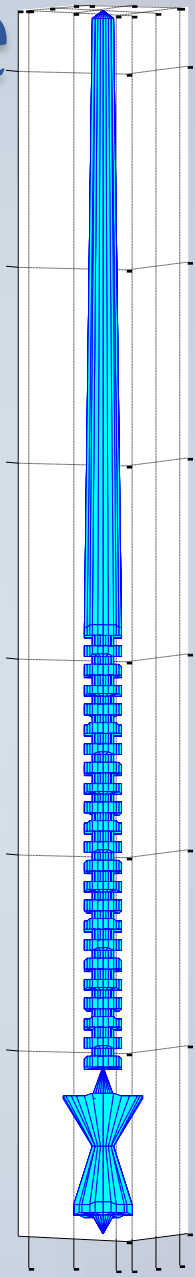
• OLD Speedup: 240x	None	10000x+	None	133x	397x
• NEW Speedup: 894x	10.3x	10000x+	~8.7x	412x	1183x
• NEW Speedup: 2423x	72x	10000x+	~10.9x	1334x	1976x
• NEW Speedup: 2925x	46x	10000x+	~9.8x	1332x	2968x

Polycone

...

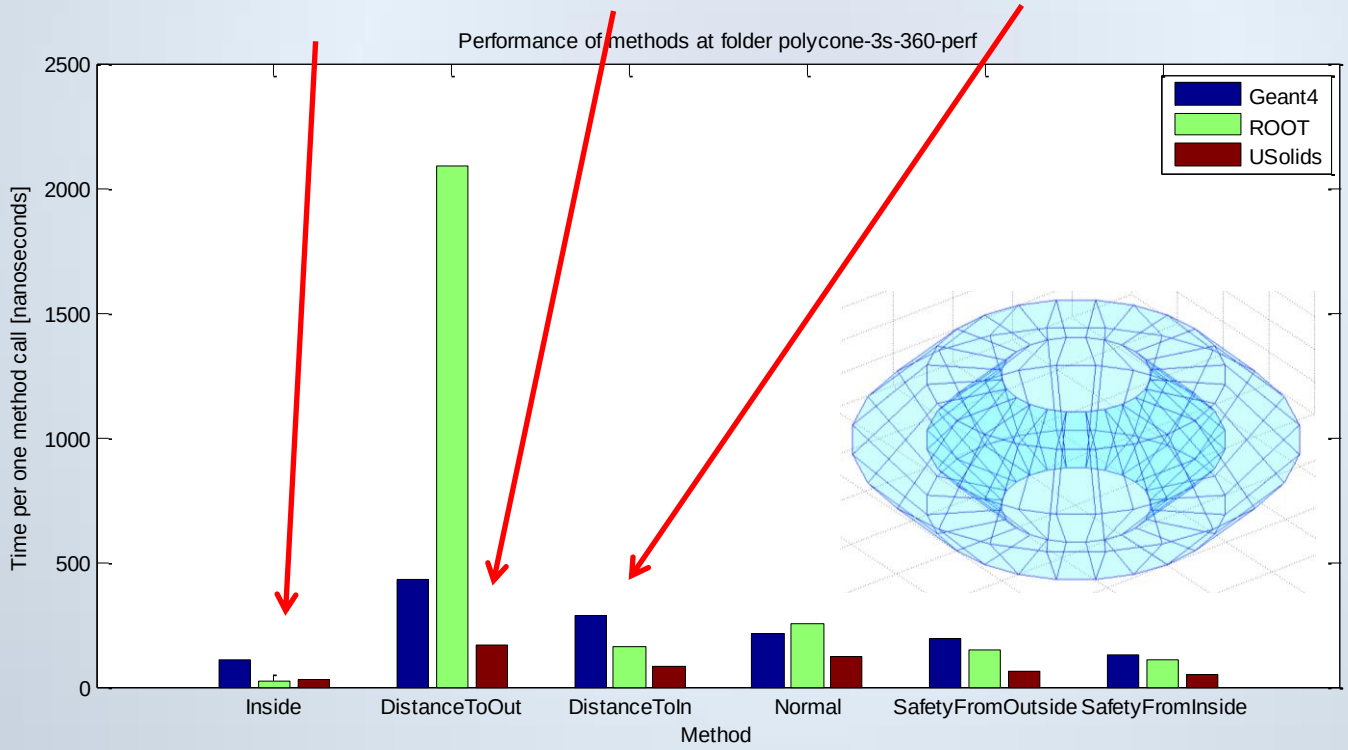
New Ordinary Polycone implementation

- Polycone is an important solid, heavily used in most experimental setups
- Special optimization for common cases: Z sections can only increase (big majority of real cases)
  - Based on composition of separate instances of cones, tubes (or their sections)
  - This way, providing brief, readable, clean and fast code
  - Excellent scalability over the number of Z sections
  - Very significant performance improvement
- Will make visible CPU improvement in full simulation of complex setups like ATLAS and CMS
  - In CMS, 6.92% was measured as the total CPU share of Polycone methods
  - Assuming possible speedup factor ~5x, this would save **around 5% total time of the whole CMS experiment**
  - Cost of polycone in ATLAS is ***"around 10%"***



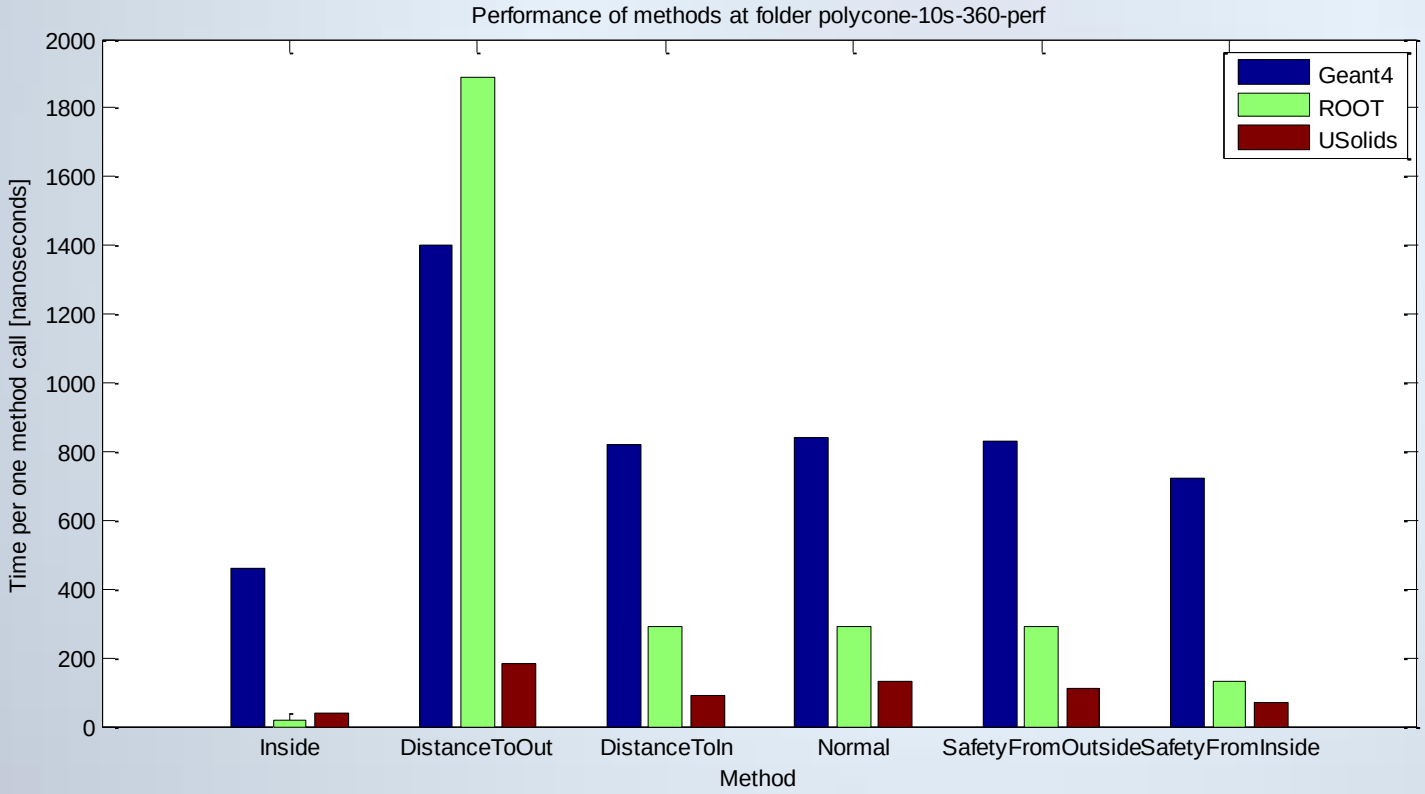
Ordinary Polycone performance, 3 z sections

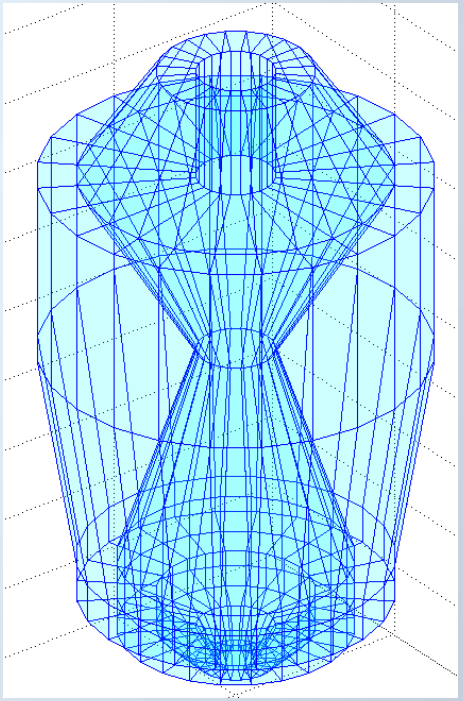
- Speedup factor **3.3x** vs. Geant4, **7.6x** vs. ROOT for most performance critical methods, i.e.: *Inside, DistanceToOut, DistanceToIn*



Ordinary Polycone performance, 10 z sections

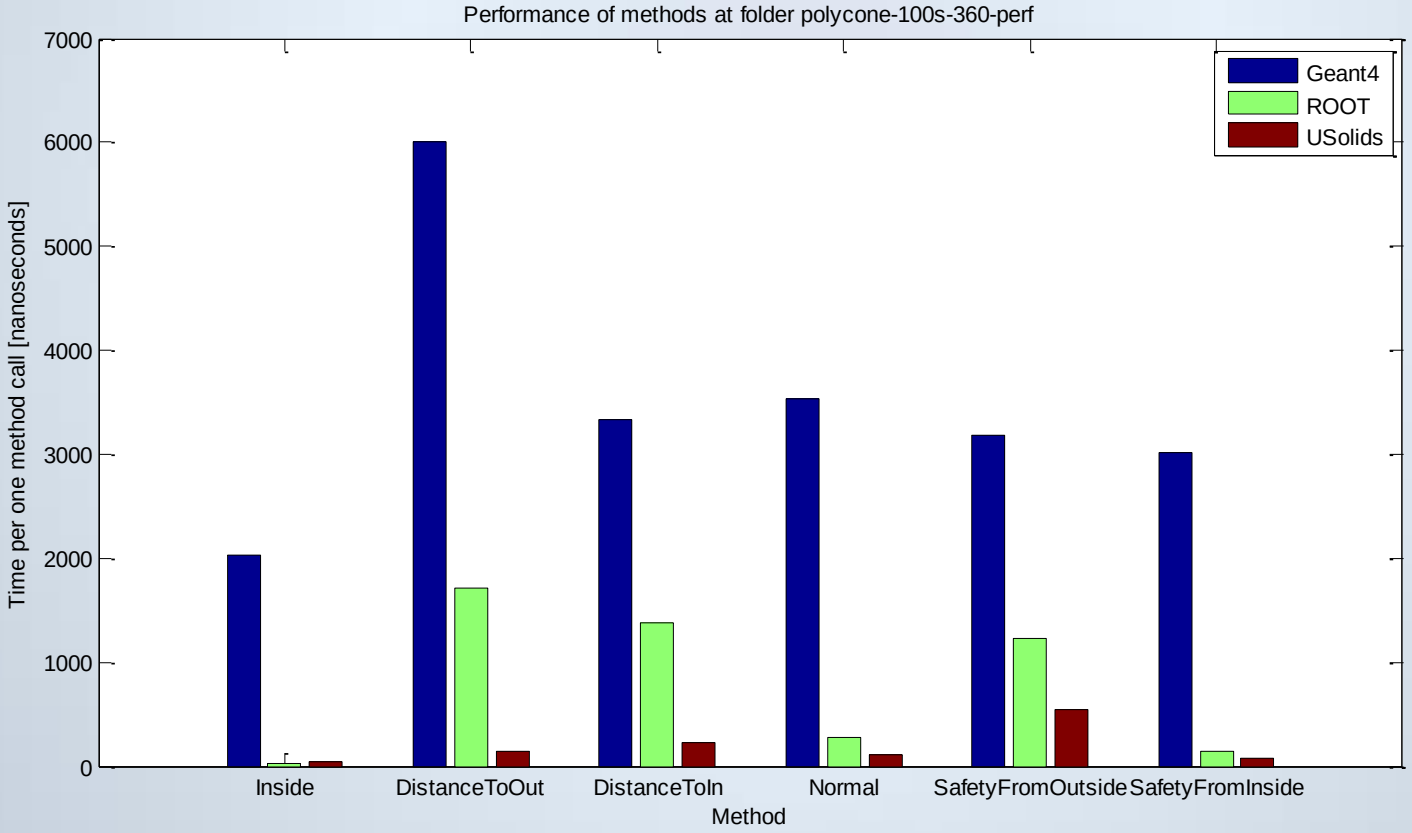
- Speedup factor **8.3x** vs. Geant4, **7.9x** vs. ROOT for most performance critical methods

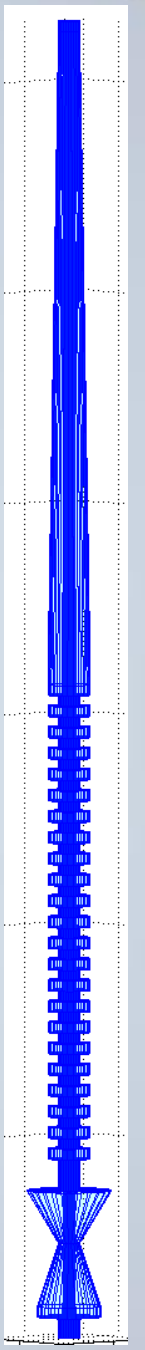




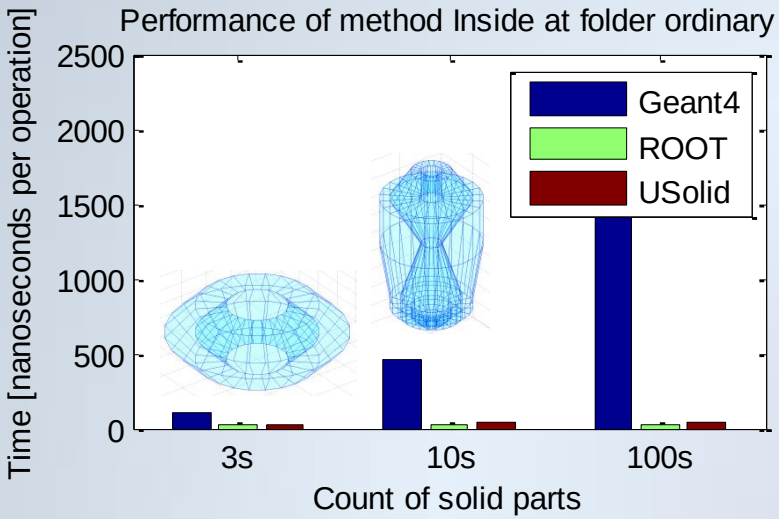
Ordinary Polycone performance, 100 z sections

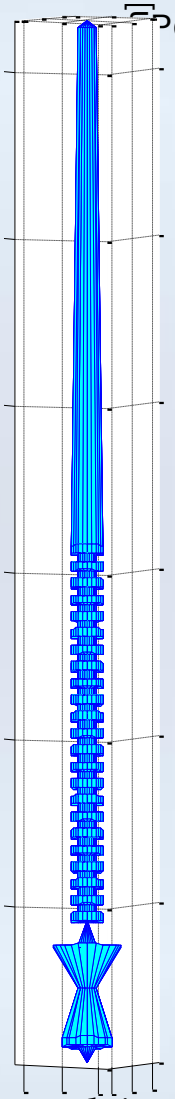
- Speedup factor **34.3x** vs. Geant4, **10.7x** vs. ROOT for most performance critical methods

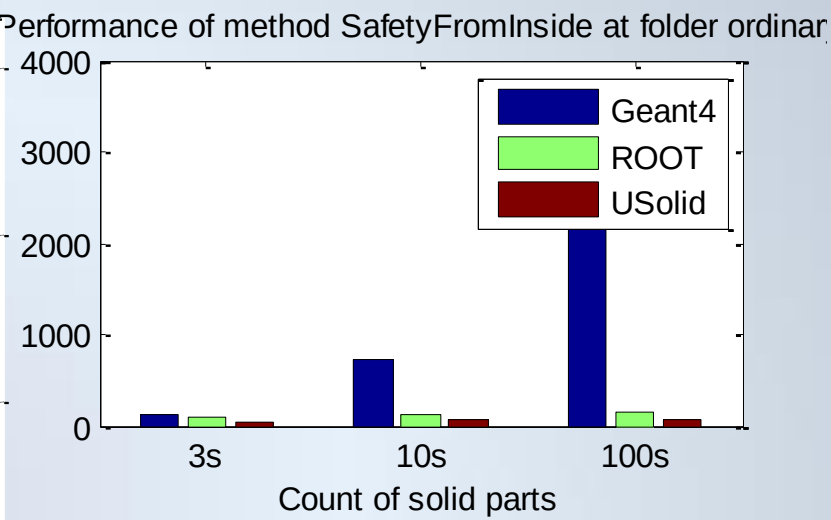


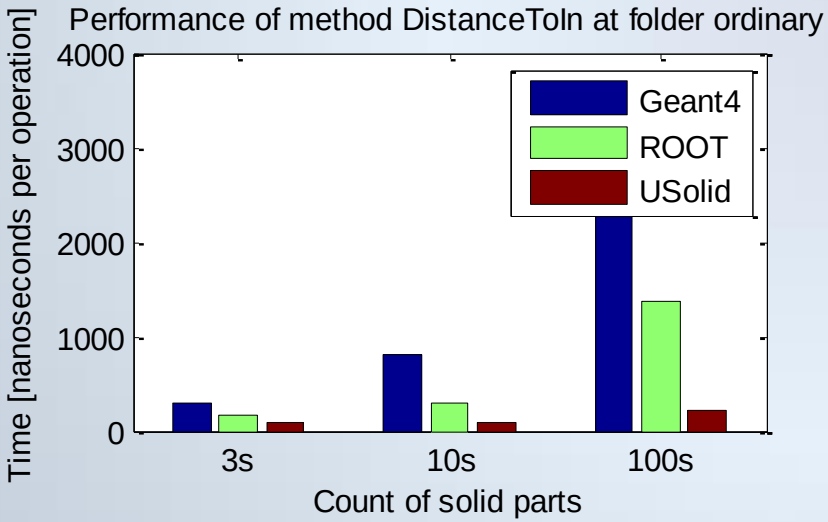


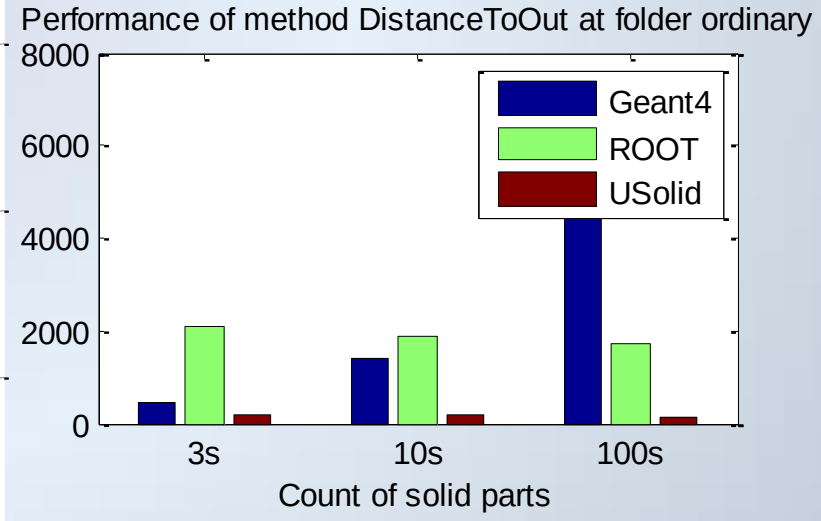
Ordinary Polycone scalability









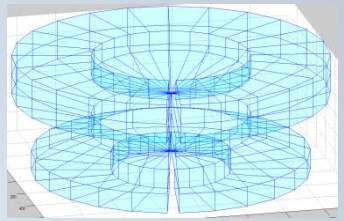


Geant4 poor performance scalability is explained by the lack of spatial optimization

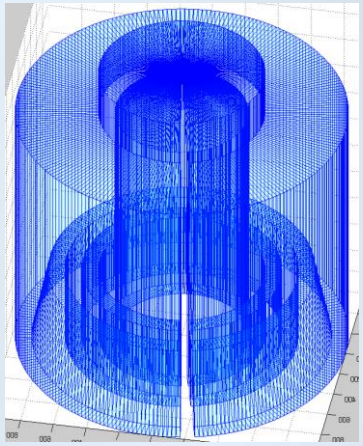
Ordinary Polycone: sometimes only proper, careful coding makes big difference

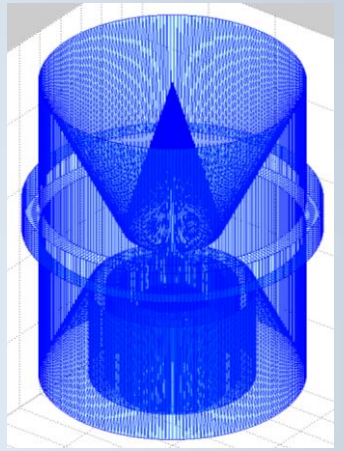
- Performance gain over Geant4 could be explained due to lack of spatial division by voxelization
- But new USolids polycone uses very similar algorithms and voxelization as ROOT does: why it is factor 7 – 10x faster, much more clear, shorter, readable and easier to maintain?
- Answer is given in 2 backup slides in the end of presentation

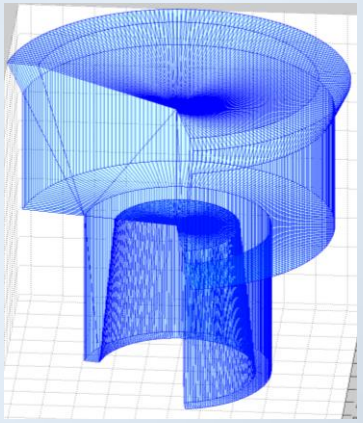
Generic Polycone

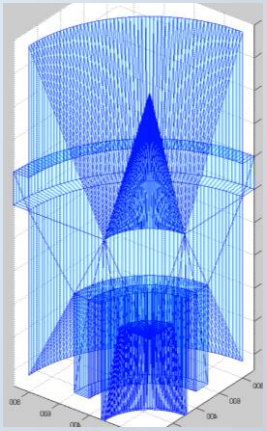


- Optimization made as well
  - Using voxelization on generalized surface facet model
  - Performance improvement over Geant4 (generic polycone does not exist in ROOT)
  - Performance improvement depends on the number of sections and is less than in case of ordinary polycone
  - Scalability is excellent as well as for the ordinary polycone case
  - Tested and measured on original Geant4 test cases of solids
  - Values are 100% conformal with Geant4
- This work was achieved thanks to the 3 month extension negotiated by John Harvey and supported by the Linear Collider Detector Team



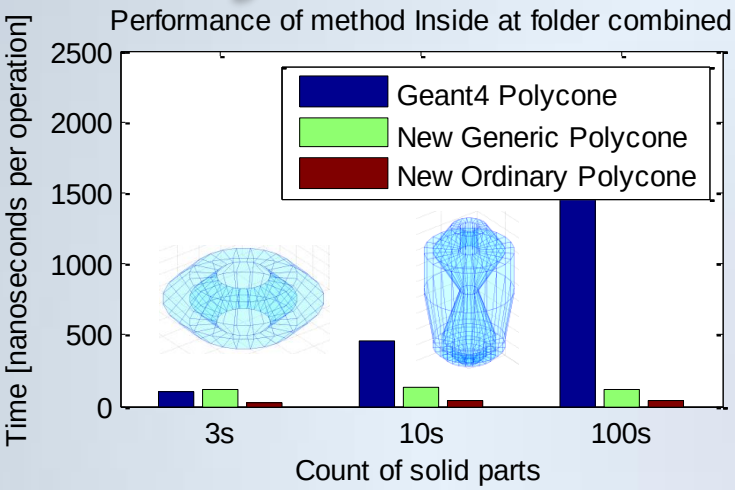


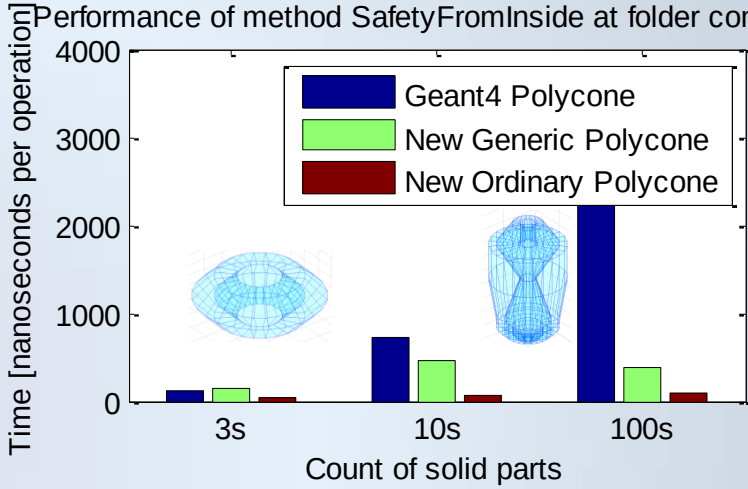


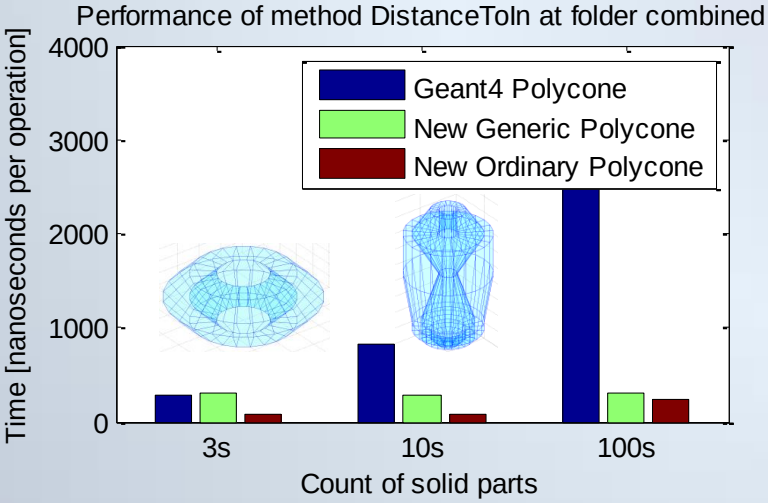


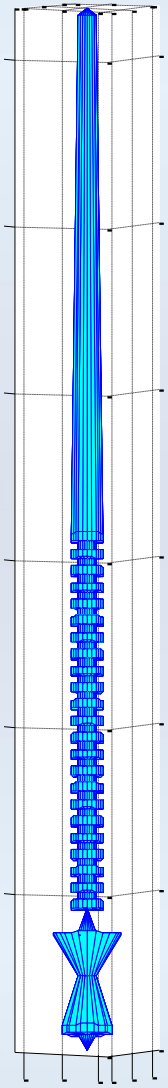
Ordinary vs. Generic Polycone

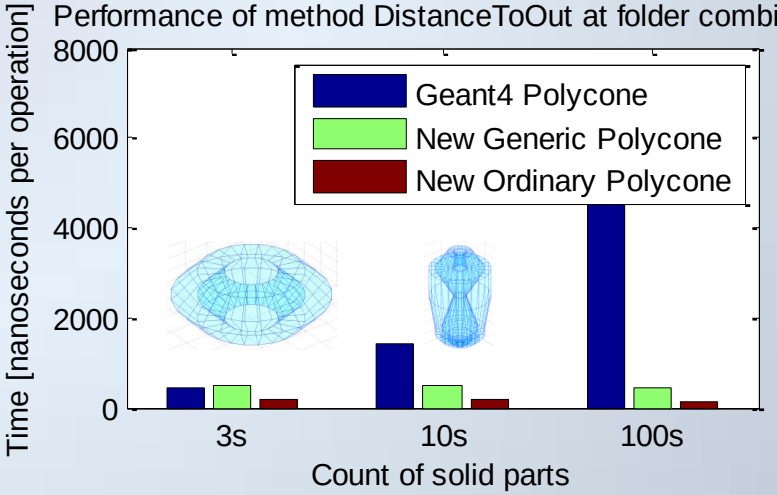
Geant4 poor performance scalability is explained by the lack of spatial optimization





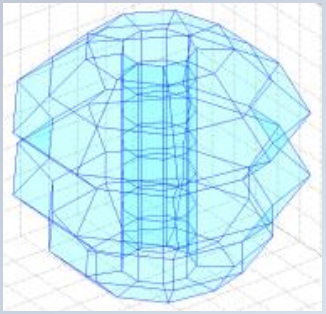






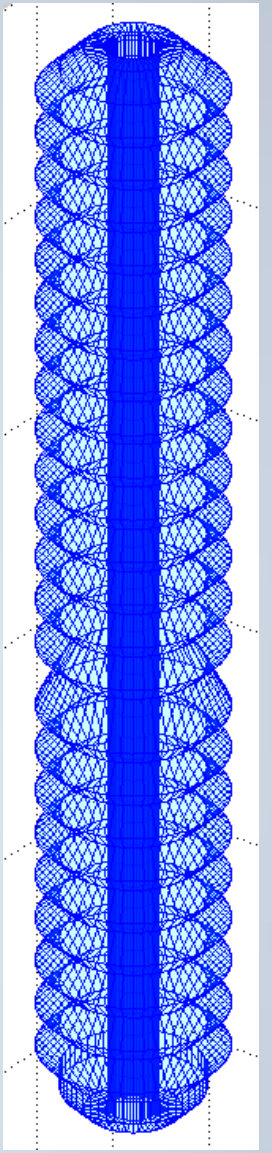
Polyhedra

...

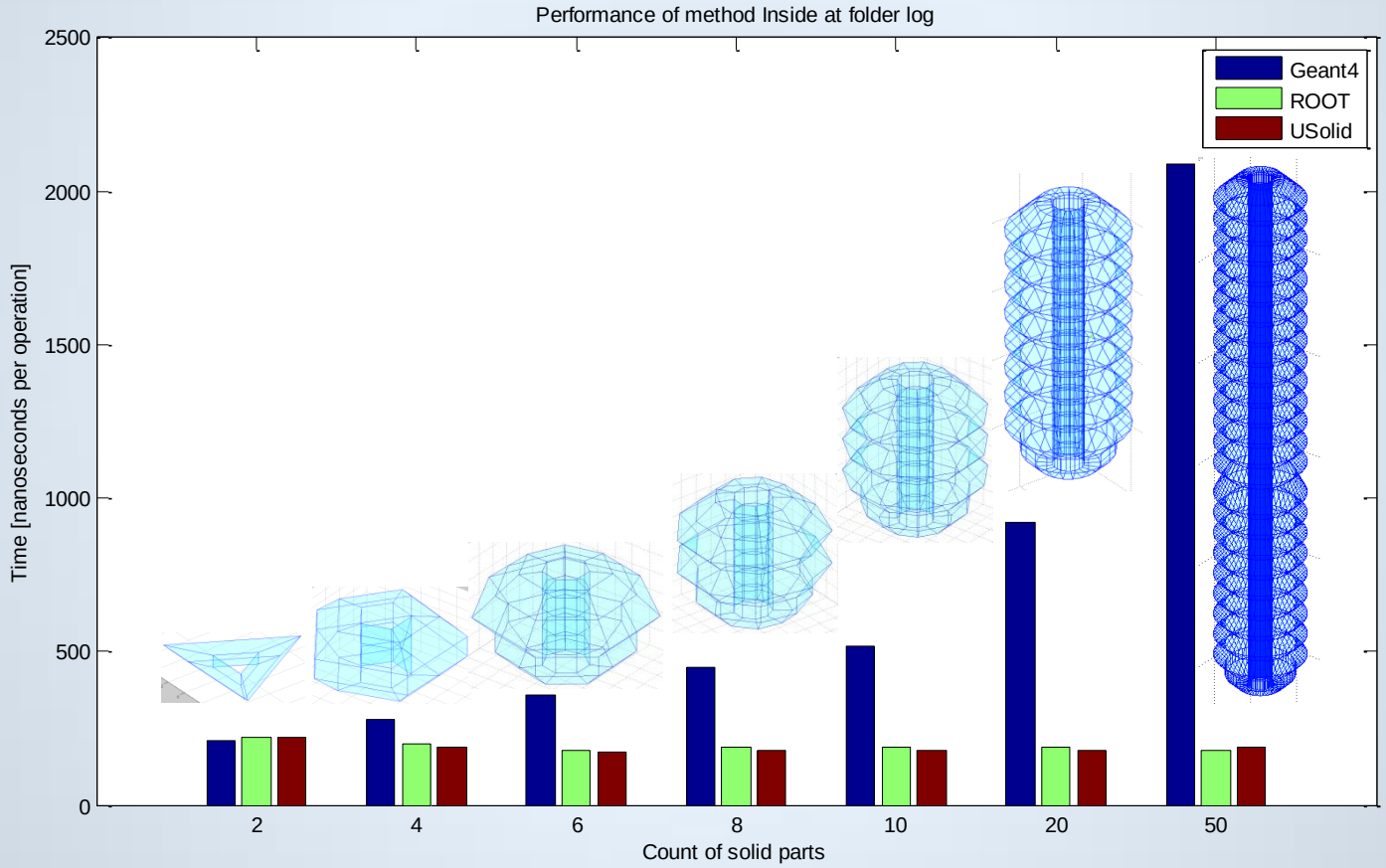


New Polyhedra implementation

- Uses similar methodology as was done for the Generic Polycone
- Works for both ordinary and generic polyhedra
- Provides improvement of performance and scalability
- Values are 100% conformal with Geant4
- Same as for the case of generic polycone, this work was made during my last 3 months contract extension

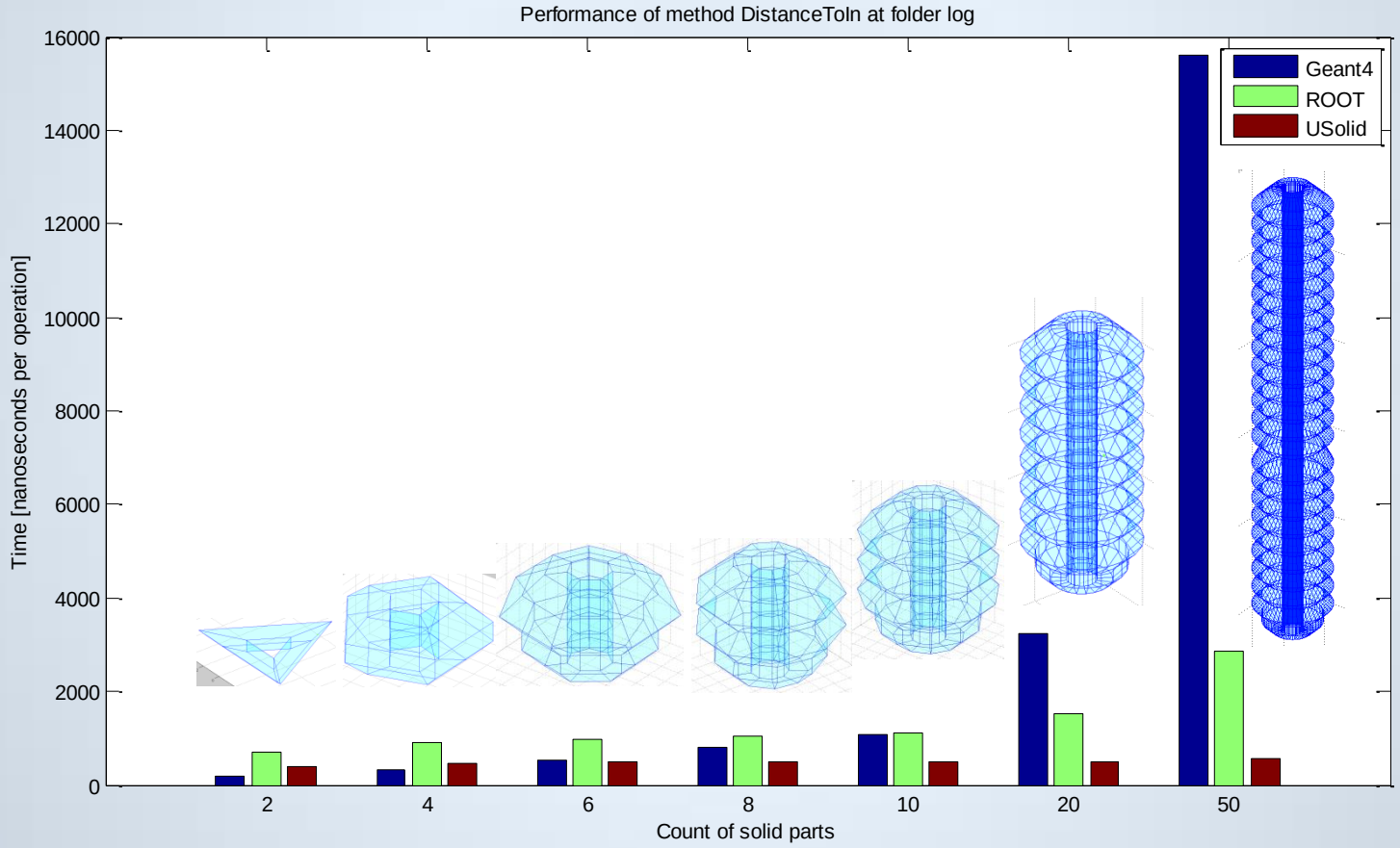


Inside scalability



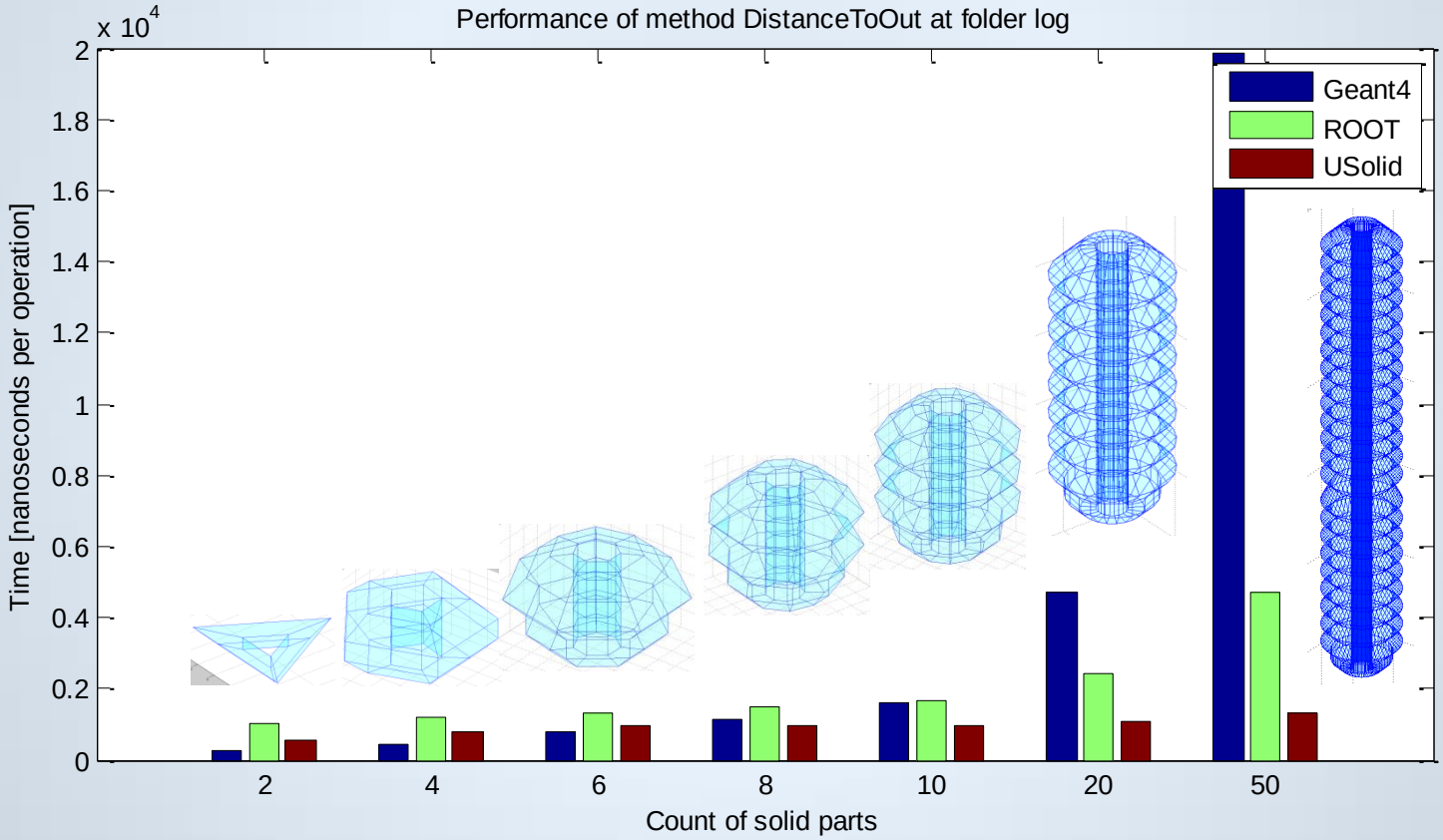
Geant4 poor performance scalability is explained by the lack of spatial optimization

DistanceToIn scalability



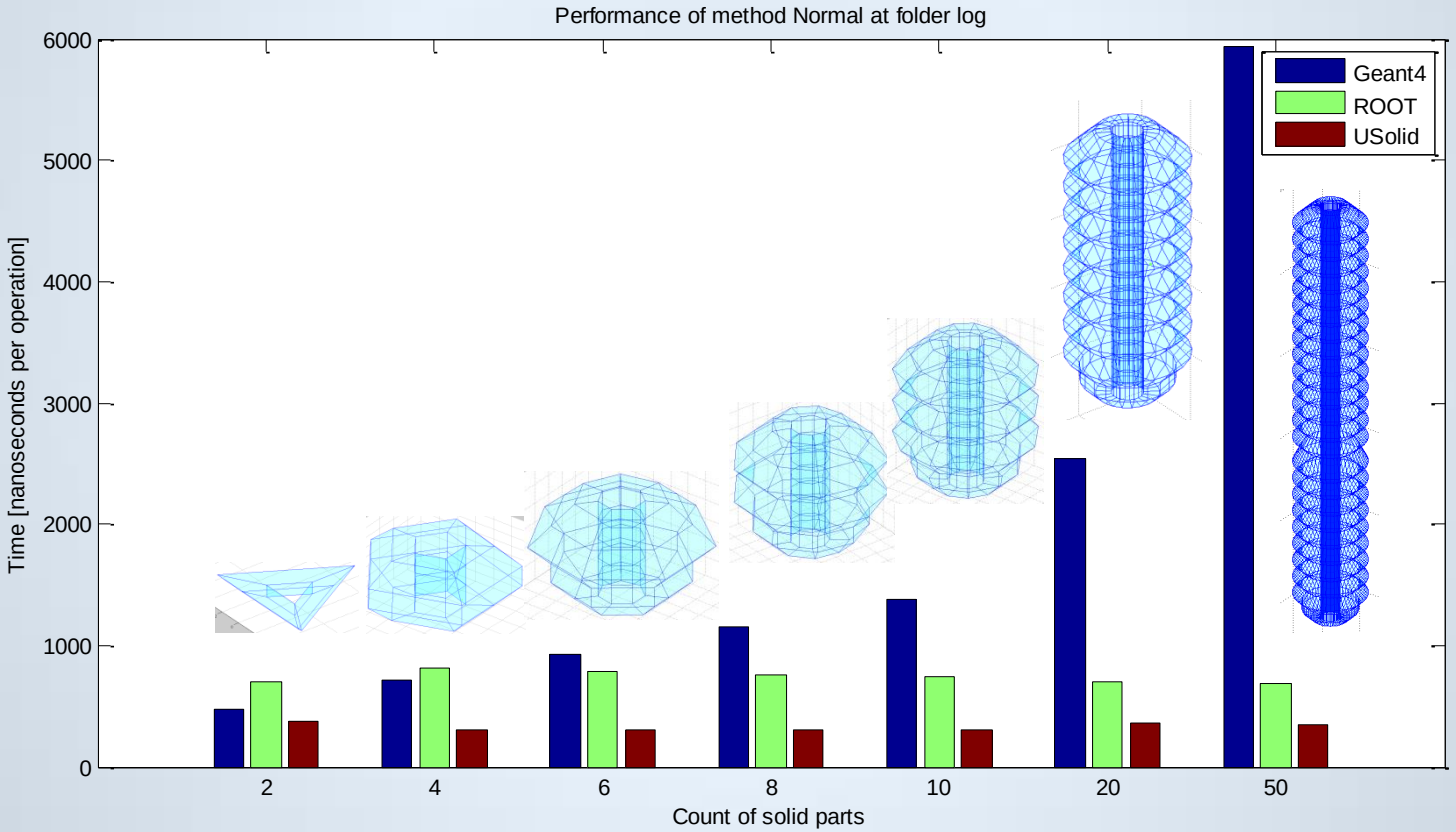
Geant4 poor performance scalability is explained by the lack of spatial optimization

DistanceToOut scalability



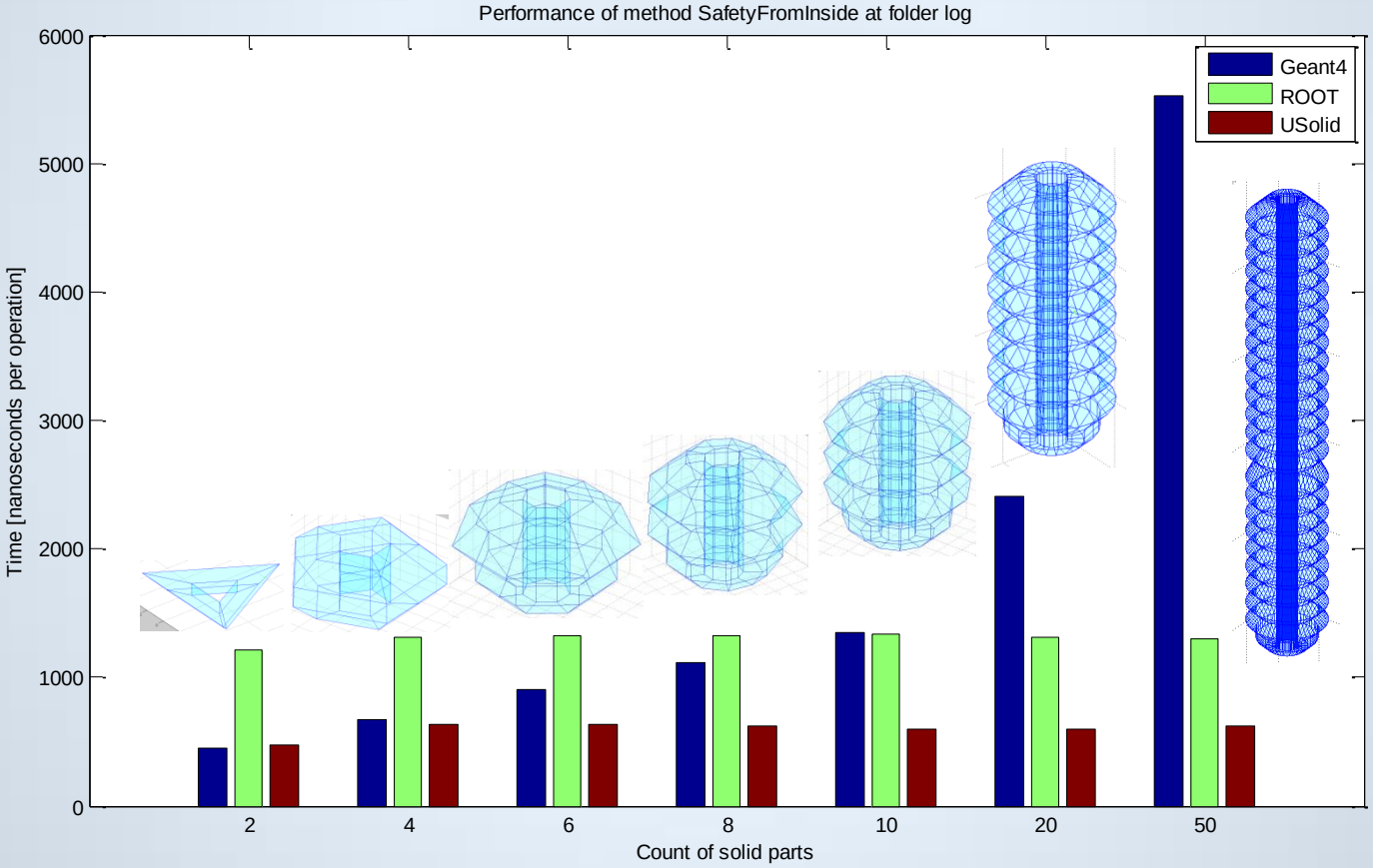
Geant4 poor performance scalability is explained by the lack of spatial optimization

Normal scalability



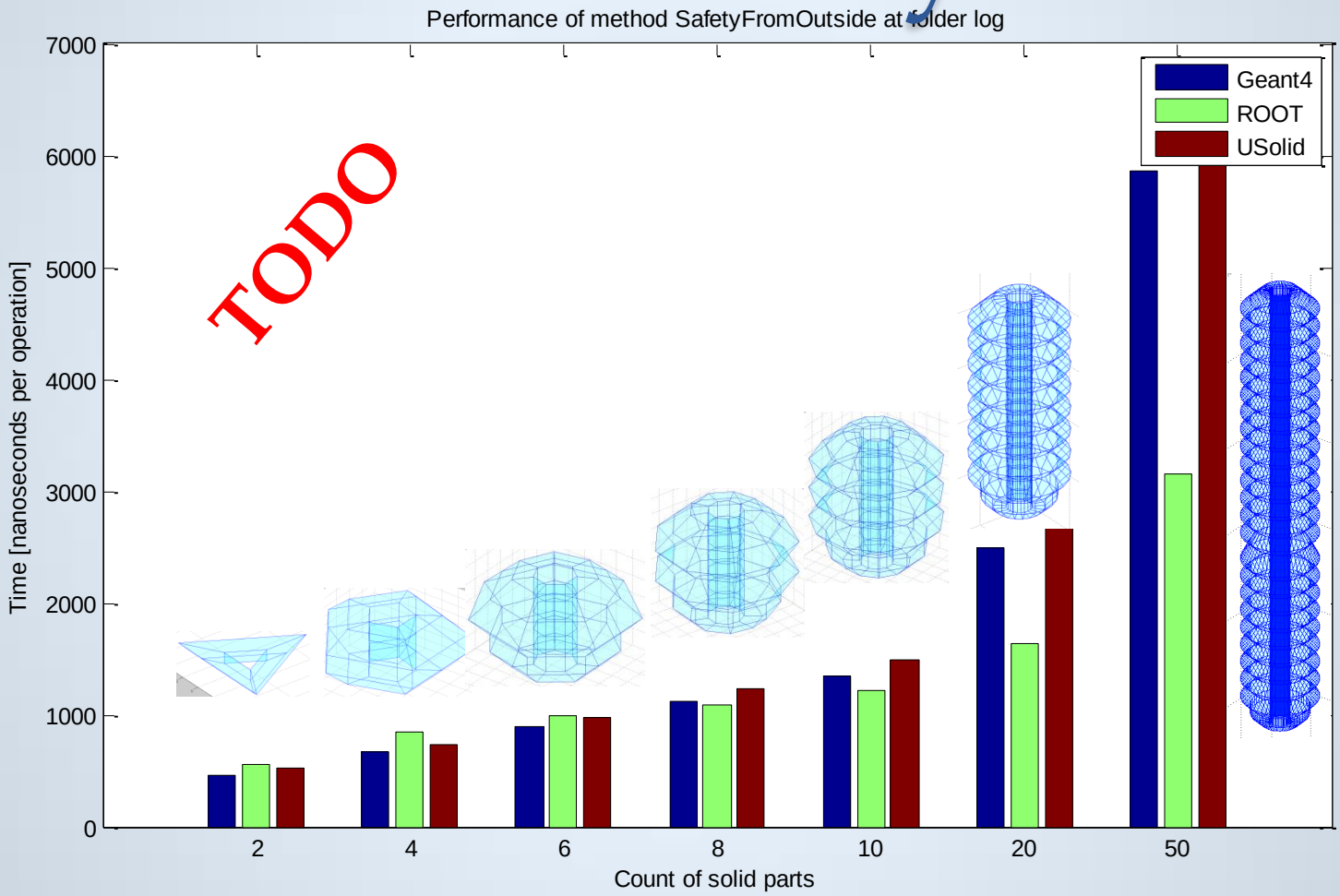
Geant4 poor performance scalability is explained by the lack of spatial optimization

SafetyFromInside scalability



Geant4 poor performance scalability is explained by the lack of spatial optimization

SafetyFromOutside scalability



Geant4 poor performance scalability is explained by the lack of spatial optimization

Status of work

- ✓ Types and Unified Solids interface are defined
- ✓ Bridge classes for comparison with Geant4 and ROOT implemented
- ✓ Testing suite defined and deployed
- ✓ Implemented 9 solid primitives
- ✓ New, high performance implementation of composed solids: Multi-Union, Tessellated solid, Polycone, Polyhedra and Extruded-Solid (ongoing)
- ✓ Code and knowledge is being passed since Q1 2013 to Tatiana Nikitina

Future work plan

- Marek Gayer
  - Finish documentation
- Tatiana Nikitina
  - Analyze and implement remaining solids for the new library
  - Verify implementation of secondary methods for all solids
  - Integration to Geant4 10

Thank you for your attention.

Questions and Answers

Backup

...

Polycone: sometimes only proper, careful coding makes huge difference 1/2

- **But new USolids polycone uses very similar algorithms and voxelization as ROOT does: why it is factor 7 – 10x faster, much more clear, shorter, readable and easier to maintain?**
- Pre-calculates as much as possible in the constructor and not during runtime, not needing to do it in navigation methods again (e.g. if it is cylindrical or tubular section)
- Using directly methods from Tubs on Cons for polycone section to separate and move the logic from Polycone to Polycone sections
  - not re-implementing it again in new methods => difficult to read, maintain
- Lot of polycone data is instead contained in Tubs and Cons

Polycone: sometimes only proper, careful coding makes huge difference 2/2

- No use of recursion, troubling compiler, processor and code readability
- *DistanceToIn* moves point to bounding box
- *std::lower_bound* is much faster than doing binary search by own coding
- C++, not feeling of mere C wrapped in classes
- Using class for vector (x,y,z) is better than using C array
- No use of pointers, memcpy, memset, …
- Section data not contained in Tubs, Cons are grouped together in a *std::vector* of struct's
- Logical components, namely repeatedly used are in separated methods, often inlined